.NET FULL STACK Development Program

—

Day 2 C# Basic

Outline

- Comment
- Variable and Data Type
- Operator
- Flow Control
- Keywords

Comment

In C#, there are 3 types of comments:

- Single-line Comments (//)
  Single-line comments start with a double slash //. The compiler ignores everything after // to the end of the line.

- Multi-line Comments (/* */)
  Multi-line comments start with /* and ends with */. Multi-line comments can span over multiple lines.

- XML Comments (///)
  XML documentation comment is a special feature in C#. It starts with a triple slash /// and is used to categorically describe a piece of code.. This is done using XML tags within a comment. These comments are then, used to create a separate XML documentation file.

```
int myInt = 100 + 1; // result is 101
```

```
/*
 * This is a multi-line comment
 * This is a multi-line comment
 * This is a multi-line comment
 */
```

```
/// <summary>
/// This is the Main method
/// The entry point of a C# application
/// </summary>
0 references
static void Main(string[] args)
{
```

Variable and Data Type

Variable

- A variable is a named memory location that we can create in order to store data.
- Every variable has a type that determines what values can be stored in the variables.

- Declaration
  [data type] [variable name];
- Initialization
  [variable name] = [value];
- Combined
  [data type] [variable name] = [value];

```C#
//declare the variable age first, initialize it later
int age;
age = 18;

//declare the variable and assign a value
int fullScore = 100;
```

Variables and Data Types

- Types of variables
  - A data type specifies the type of data that a variable can store.
  - Two kinds of types:
    - Value types:
      simple types, *enum types*, *struct types*, *nullable value types*, and *tuple value types.*
    - *Reference types:*
      string, class types, *interface types*, *array types*, and *delegate types.*

Value Type

- Simple Types
  - Signed integral: sbyte, short, int, long
  - Unsigned integral: byte, ushort, uint, ulong
  - floating-point: float, double
  - High-precision decimal floating-point: decimal
  - Unicode characters: char
  - Boolean: bool

Signed and Unsigned Integral

C# supports the following predefined integral types:

C# type/keyword	Range	Size	.NET type
`sbyte`	-128 to 127	Signed 8-bit integer	System.SByte
`byte`	0 to 255	Unsigned 8-bit integer	System.Byte
`short`	-32,768 to 32,767	Signed 16-bit integer	System.Int16
`ushort`	0 to 65,535	Unsigned 16-bit integer	System.UInt16
`int`	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	System.Int32
`uint`	0 to 4,294,967,295	Unsigned 32-bit integer	System.UInt32
`long`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	System.Int64
`ulong`	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	System.UInt64

nint & nuint

- Starting in C# 9.0, you can use the nint and nuint keywords to define ***native-sized integers***. These are 32-bit integers when running in a 32-bit process, or 64-bit integers when running in a 64-bit process.
- The native-sized integer types are represented internally as the .NET types System.IntPtr and System.UIntPtr.
- Range:
  - For **nint**: Int32.MinValue to Int32.MaxValue.
  - For **nuint**: UInt32.MinValue to UInt32.MaxValue.
- There’s no direct syntax for native-sized integer literals. There’s no suffix to indicate that a literal is a native-sized integer, such as L/l to indicate a long. You can use implicit or explicit casts of other integer values instead.

```
nint nativeSignedInt = 10;
nuint nativeUnsignedInt = (nuint)11;
Console.WriteLine(nativeSignedInt.GetType()); // System.IntPtr
Console.WriteLine(nativeUnsignedInt.GetType());//System.UIntPtr
```

Floating-point numeric types

C# type/keyword	Approximate range	Precision	Size	.NET type
`float`	±1.5 x 10^{-45} to ±3.4 x 10^{38}	~6-9 digits	4 bytes	System.Single
`double`	±5.0 × 10^{-324} to ±1.7 × 10^{308}	~15-17 digits	8 bytes	System.Double
`decimal`	±1.0 x 10^{-28} to ±7.9228 x 10^{28}	28-29 digits	16 bytes	System.Decimal

```
//to declare floating-point numbers
float myFloat = 11.1f;
double myDouble = 11.1;
decimal myDecimal = 11.1m;
```

Unicode Character: char

The **char** type keyword represents a Unicode UTF-16 character.

Type	Range	Size	.NET type
`char`	U+0000 to U+FFFF	16 bit	System.Char

```
char myChar = 'a';
myChar = '@';
```

Dec	Char	Dec	Char	Dec	Char	Dec	Char
0	NUL (null)	32	SPACE	64	@	96	`
1	SOH (start of heading)	33	!	65	A	97	a
2	STX (start of text)	34	"	66	B	98	b
3	ETX (end of text)	35	#	67	C	99	c
4	EOT (end of transmission)	36	$	68	D	100	d
5	ENQ (enquiry)	37	%	69	E	101	e
6	ACK (acknowledge)	38	&	70	F	102	f
7	BEL (bell)	39	'	71	G	103	g
8	BS (backspace)	40	(	72	H	104	h
9	TAB (horizontal tab)	41	)	73	I	105	i
10	LF (NL line feed, new line)	42	*	74	J	106	j
11	VT (vertical tab)	43	+	75	K	107	k
12	FF (NP form feed, new page)	44	,	76	L	108	l
13	CR (carriage return)	45	-	77	M	109	m
14	SO (shift out)	46	.	78	N	110	n
15	SI (shift in)	47	/	79	O	111	o
16	DLE (data link escape)	48	0	80	P	112	p
17	DC1 (device control 1)	49	1	81	Q	113	q
18	DC2 (device control 2)	50	2	82	R	114	r
19	DC3 (device control 3)	51	3	83	S	115	s
20	DC4 (device control 4)	52	4	84	T	116	t
21	NAK (negative acknowledge)	53	5	85	U	117	u
22	SYN (synchronous idle)	54	6	86	V	118	v
23	ETB (end of trans. block)	55	7	87	W	119	w
24	CAN (cancel)	56	8	88	X	120	x
25	EM (end of medium)	57	9	89	Y	121	y
26	SUB (substitute)	58	:	90	Z	122	z
27	ESC (escape)	59	;	91	[	123	{
28	FS (file separator)	60	<	92	\	124	|
29	GS (group separator)	61	=	93	]	125	}
30	RS (record separator)	62	>	94	^	126	~
31	US (unit separator)	63	?	95	_	127	DEL

Type Conversion and Casting

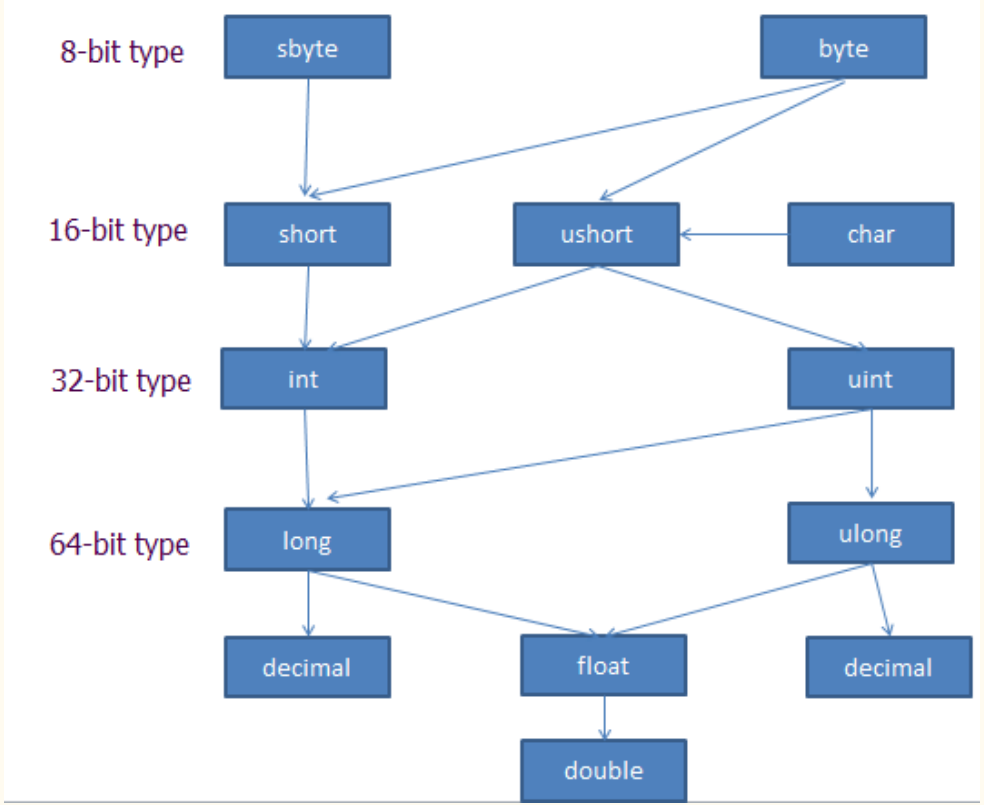
- **Implicit conversions**: No special syntax is required because the conversion always succeeds, and "no data will be lost". Examples include conversions from smaller to larger integral types.

  - **Implicit Casting** (automatically) - converting a smaller type to a larger type size
    `char` -> `int` -> `long` -> `float` -> `double`

- **Explicit conversions (casting)**: Explicit conversions require a cast expression. Casting is required when information might be lost in the conversion, or when the conversion might not succeed for other reasons. Typical examples include numeric conversion to a type that has less precision or a smaller range.

  - **Explicit Casting** (manually) - converting a larger type to a smaller size type
    `double` -> `float` -> `long` -> `int` -> `char`

Type Conversion and Casting



Boolean

The **bool** type keyword represents a Boolean value, which can be either true or false.

bool	1 bit	Stores true or false values

C#

```
bool itemAvailability = true;

itemAvailability = false;
```

Value Types

- Value types
  - Simple types
    - Signed integral: sbyte, short, int, long
    - Unsigned integral: byte, ushort, uint, ulong
    - floating-point: float, double
    - High-precision decimal floating-point: decimal
    - Unicode characters: char
    - Boolean: bool
  - Enum types
    - User-defined types of the form enum E {...}. An enum type is a distinct type with named constants. Every enum type has an underlying type, which must be one of the eight integral types. The set of values of an enum type is the same as the set of values of the underlying type.
  - Struct types
    - User-defined types of the form struct S {...}
  - Nullable value types
    - Extensions of all other value types with a null value
  - Tuple value types
    - User-defined types of the form (T1, T2, ...)

Enumeration type

An ***enumeration type*** (or *enum type*) is a value type. To define an enumeration type, use the ***enum*** keyword and specify the names of *enum members*

```
enum Week
{
    Monday,    // 0
    Tuesday,   // 1
    Wednesday, // 2
    Thursday,  // 3
    Friday,    // 4
    Saturday,  // 5
    Sunday     // 6
}
0 references
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");
    //an enum value can be access by using the [enum_name].[member_name]
    Week week = Week.Monday;
    Console.WriteLine(week);

    //we can also access thru the index
    int enumIndex = (int)Week.Sunday;
    Console.WriteLine(enumIndex); // 6

    Console.WriteLine((Week)4); // Friday
}
```

Structure type

- A structure type (or struct type) is a value type that can encapsulate data and related functionality. You use the struct keyword to define a structure type.
- We can have variables, methods, and constructors defined in the struct.
- Struct doesn’t allow no-arg constructors.

```
3 references
struct Album
{
    string _title;
    string _singer;
    string _date;

    1 reference
    public Album(string title, string singer, string date)
    {
        _title = title;
        _singer = singer;
        _date = date;
    }

    1 reference
    public void GetReleaseDate ()
    {
        Console.WriteLine(_date);
    }
}
0 references
static void Main(string[] args)
{
    // we need to use new keyword to create an album instance
    Album album = new Album("Recovery", "Eminem", "2010-06-18");
    //call the method in the struct and write the release date in the console
    album.GetReleaseDate(); // 2010-06-18
}
```

Tuple type

- Available in C# 7.0 and later, tuple types are value types, the tuples feature provides concise syntax to group multiple data elements in a lightweight data structure.

```
(string, int) t1 = ("Liam", 22);
(string name, int age) t2 = ("Lily", 23);
Console.WriteLine("Name is " + t1.Item1 + ", " + "Age is " + t1.Item2);
Console.WriteLine($"Name is {t2.name}, Age is {t2.age}");
```

Nullable Value Type - ?

- Starting from C# 8.0, A nullable value type represents all values of its underlying value type and an additional null value. For example, you can assign any of the following three values to a ***bool?*** variable: **true**, **false**, or **null**.

```
int? num = null;
int num2 = num; // cannot compile

char? character = 'a';
character = null;
```

- The null-coalescing operator ?? returns the value of its left-hand operand if it isn't null; otherwise, it evaluates the right-hand operand and returns its result. The ?? Operator doesn't evaluate its right-hand operand if the left-hand operand evaluates to non-null.

```
int? a = 20;
int? b = a +10;
Console.WriteLine("a = " + a + " and b =" + b);
a = null;
b = a + 10 ?? -1; // if a + 10 is null, assign -1 to b
Console.WriteLine("a = " + a + " and b =" + b);
```

```
a = 20 and b =30
a =  and b =-1
```

Reference Types

- C# provides the following built-in reference types:
  - string
  - dynamic
  - object

- The following keywords are used to declare reference types:
  - class
  - delegate
  - interface
  - record

String type

The **string** type represents a sequence of zero or more Unicode characters (char).

```
string sayHi = "Hello December!";
//we can use index to access every char in the string
char oneChar = sayHi[0];
Console.WriteLine(oneChar); // H
```

Immutability of Strings

String objects are ***immutable***: they can't be changed after they've been created.

When we assign a value to a string or modify the string, C# will create a new string.

```
Console.WriteLine("String Immutability:");
string herName = "Sophia";
herName.ToUpper(); // Cap all the letters in the string
Console.WriteLine(herName);
```

String Interning

- The common language runtime(CLR) conserves string storage by maintaining a table, called the ***intern pool***, that contains a single reference to each unique literal string declared or created programmatically in your program. Consequently, an instance of a literal string with a particular value only exists once in the system.

- For example, if you assign the same literal string to several variables, the runtime retrieves the same reference to the literal string from the intern pool and assigns it to each variable.

String Concatenation

```
// string concatenation -> +
string str1 = "New";
string str2 = "Jersey";
string combinedString1 = str1 + str2;
Console.WriteLine(combinedString1); // NewJersey

//string concatenation -> Concat()
string combinedString2 = string.Concat(str1, str2);
Console.WriteLine(combinedString2); // NewJersey

//string interpolation
string stringInterpolation = $"{combinedString2} is the garden state.";
Console.WriteLine(stringInterpolation);

//string formatting
(string name, int age) myInfo = ("Liam", 22);
Console.WriteLine("Hello, I am {0}, {1} years old.", myInfo.name, myInfo.age);
```

- string.Format();
- string.Join();
- Append() method in StringBuilder;

Class

A class is a container that contains the block of code that includes field, method, constructor, etc.

```
class Child
{
    private int age;
    private string name;

    // Default constructor:
    public Child()
    {
        name = "N/A";
    }

    // Constructor:
    public Child(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    // Printing method:
    public void PrintChild()
    {
        Console.WriteLine("{0}, {1} years old.", name, age);
    }
}
```

Delegate

A **delegate** is a reference type that can be used to encapsulate a named or an anonymous method. The delegate must be instantiated with a method or lambda expression that has a compatible return type and input parameters.

```
//Define Delegate
public delegate void Hello();

0 references
public static void Main(string[] args)
{
    //approach 1
    Hello delegate1 = Function;
    //approach 2
    Hello delegate2 = delegate () { Console.Write("Hi2"); };
    //approach 3 - lambda
    Hello delegate3 = () => { Console.Write("Hi3"); };
    //approach 4
    Hello delegate4 = new Hello(Function);
    //approach 5
    Hello delegate5 = delegate1+ delegate2+ delegate3+ delegate4;

    //invoke delegate function
    delegate5();

}

2 references
public static void Function() { Console.Write("Hi"); }
```

Dynamic type

The **dynamic** type indicates that the use of the variable and references to its members **escapes compile-time type checking.** Instead, it resolves the type at run time.

The **dynamic** types change types at run-time based on the assigned value, so **the type dynamic only exists at compile-time, not run-time**.

The dynamic type variables are converted to other types implicitly.

```
Console.WriteLine("dynamic types change types at run-time!");

dynamic MyDynamicVar = 100;
Console.WriteLine("Value: {0}, Type: {1}", MyDynamicVar, MyDynamicVar.GetType());

MyDynamicVar = "Hello World!";
Console.WriteLine("Value: {0}, Type: {1}", MyDynamicVar, MyDynamicVar.GetType());

MyDynamicVar = true;
Console.WriteLine("Value: {0}, Type: {1}", MyDynamicVar, MyDynamicVar.GetType());
```

```
dynamic d1 = 100;
int i = d1;

d1 = "Hello";
string greet = d1;
```

Other Data Types

Implicit type - Var

Beginning with C# 3, variables that are declared at method scope can have an implicit "type" **var**. An implicitly typed local variable is strongly typed just as if you had declared the type yourself, but the compiler determines the type.

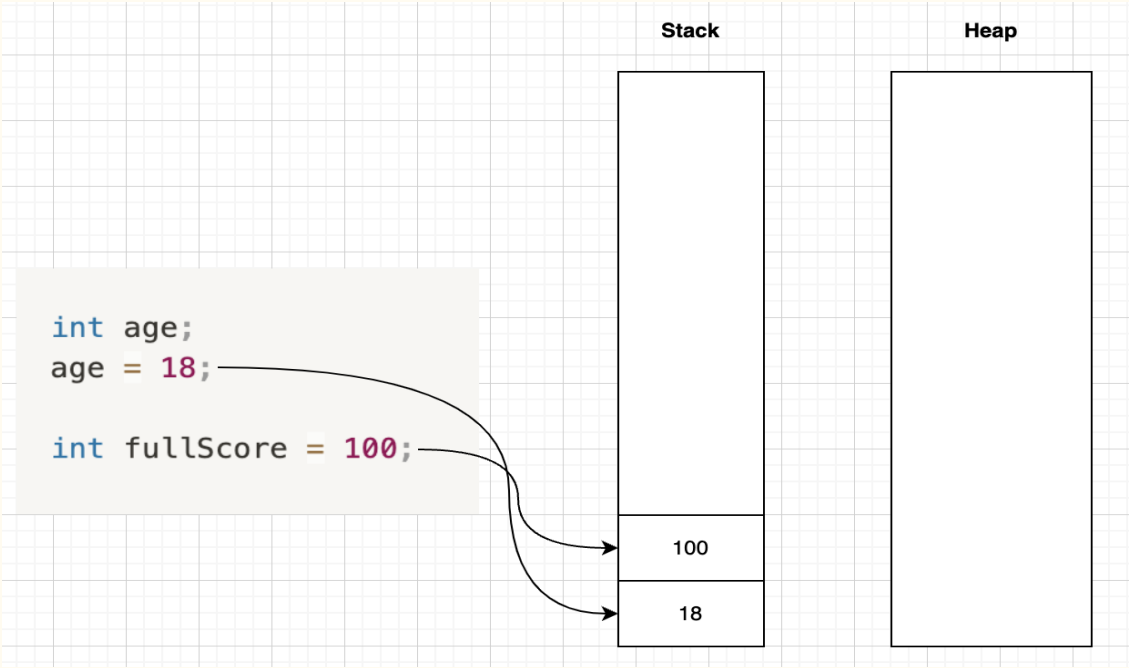
```C#
int age; //explicitly typed
age = 18;

var fullScore = 100; //implicitly typed

var month = "November"; //implicitly typed
```

Var vs. Dynamic

var vs. dynamic	var	dynamic
Type	var variables are identified at **compile time**.	dynamic variables are identified at **run time**.
Declaration	var variables **must be initialized** at the time of their declaration.	dynamic variables are **not mandatory to get initialized** at the time of declaration.
Value	var variables are **statically typed** and they cannot be assigned different data type values.	dynamic variables are **dynamically typed**, and they can be assigned different data type values.

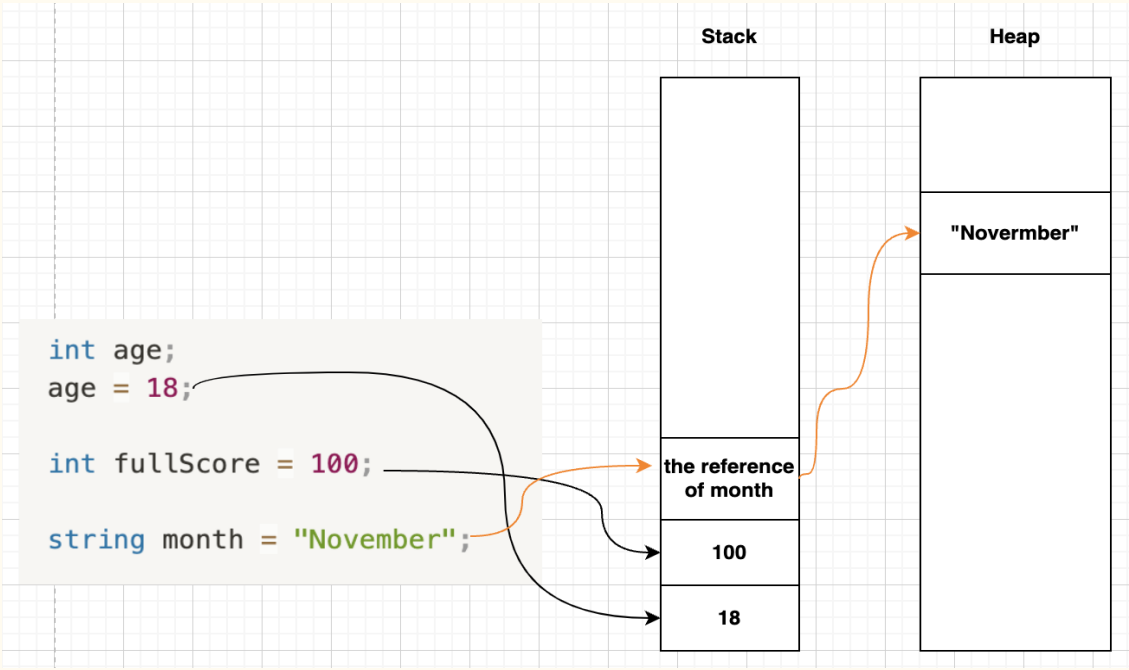
Stack

- Used for static memory allocation
- Contains data for value types
- Contains references to reference types(objects)
- Relatively small
- Access speed is fast



Heap

- Used for dynamic memory allocation
- Contains the data of reference types
- Relatively large
- Access speed is slow



Operator

Operator

Arithmetic operations in C#

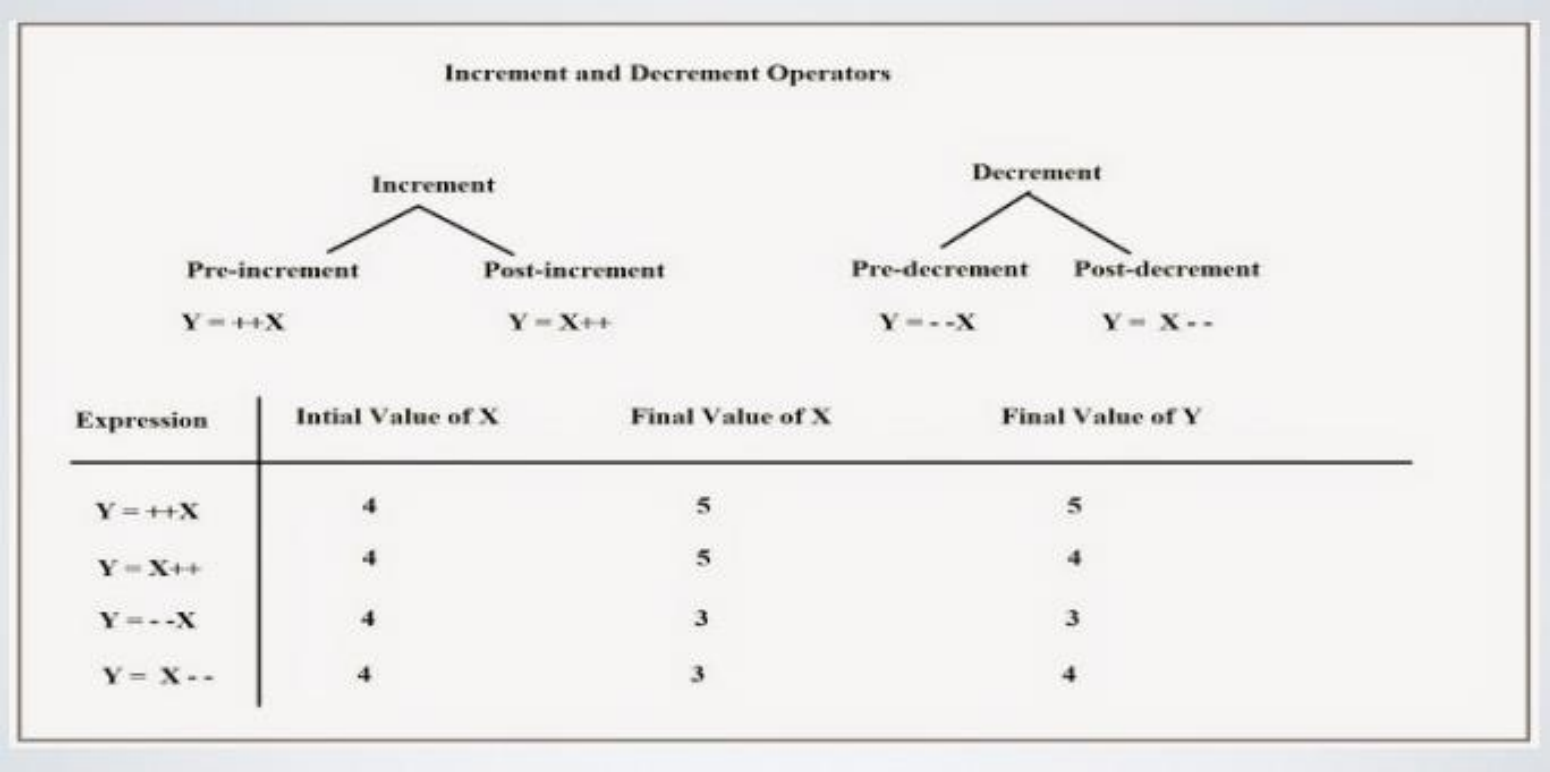
- Precedence: (*, /, %) > (+, -)
- Parentheses: evaluate the innermost parenthesized expression first, and work your way out through the levels of nesting
- No {} or [ ] in parentheses in C#

```C#
int x = 1, y = 2, z;

z = x + y * 2; // 5

z = (x + y) * 2 //6
```

Compound Arithmetic/ Assignment Operators

Operator	Use	Meaning
+=	`x += 1;`	`x = x + 1;`
-=	`x -= 1;`	`x = x - 1;`
*=	`x *= 5;`	`x = x * 5;`
/=	`x /= 2;`	`x = x / 2;`
%=	`x %= 10;`	`x = x % 10;`

Increment and Decrement Operators



Expression	Intial Value of X	Final Value of X	Final Value of Y
Y = ++X	4	5	5
Y = X++	4	5	4
Y = - -X	4	3	3
Y = X - -	4	3	4

Logical Operator

Operator	Name	Type	Description
!	Not	Unary	Returns true if the operand to the right evaluates to false. Returns false if the operand to the right is true.
&	And	Binary	Binary operation: logical and 0001 & 0011 = 0001 = 1
|	Or	Binary	Binary operation: logical or 0001 | 0011 = 0011 = 3
^	Xor	Binary	Binary Operation: exclusive or 0001 ^ 0011 = 0010 = 2
&&	Conditional And	Binary	If the operand on the left returns false, it returns false without evaluating the operand on the right.
||	Conditional Or	Binary	If the operand on the left returns true, it returns true without evaluating the operand on the right.

Relational Operators

Operator	Result
==	Equal to
!=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Flow Control

- Selection Statements
  - if, switch
- Iteration Statements
  - for, foreach, do-while, while
- Jump Statements
  - break, continue, return, goto

If

```
int time = 22;

if(time < 10)
{
    Console.WriteLine("Good Morning!");
}else if(time < 20) //when 10 <= time < 20
{
    Console.WriteLine("Good Day!");
}
else //when time >= 20
{
    Console.WriteLine("Good Evening!");
}
```

Ternary Operators

variable = condition ? expressionTrue : expressionFalse;

```
bool valid = true;
int i;

if(valid)
{
   i = 1;
} else
{
   i = 0;
}
```

```
//Ternary Operators
bool valid = true;
int i;
i = valid ? 1 : 0;
```

Switch

```
char letter = 'A';
switch (letter)
{
    case 'A':
        Console.WriteLine("Letter A");
            break;

    case 'B':
        Console.WriteLine("Letter B");
        break;

    default:
        Console.WriteLine("A Letter");
        break;
}
```

```
double measurement = 22.5;

switch (measurement)
{
    // (Introduced in C# 9.0)
    case < 0.0:
        Console.WriteLine($"Measured value is {measurement}; too low.");
        break;

    case > 15.0:
        Console.WriteLine($"Measured value is {measurement}; too high.");
        break;

    default:
        Console.WriteLine($"Measured value is {measurement}.");
        break;
}
```

Iteration Statement

- For
- Foreach
- While
- Do-while

For

```
for (initialization; condition; iterator)
{
  // statements to keep executing while condition is true
}
```

```
for (int i = 0; i < 3; i++)
{
    Console.Write(i);
}
// Output:
// 012
```

Foreach

```
foreach (element in iterable-item)
{
    // body of foreach loop
}
```

```
var fibNumbers = new List<int> { 0, 1, 1, 2, 3, 5, 8, 13 };
foreach (int element in fibNumbers)
{
    Console.Write($"{element} ");
}
// Output:
// 0 1 1 2 3 5 8 13
```

While

```
while (condition)
{
  // body of while
}
```

```
int n = 0;
while (n < 5)
{
    Console.Write(n);
    n++;
}
// Output:
// 01234
```

Do-while

```
do
{

  //body of do-while loop

} while (condition)
```

```
int n = 0;
do
{
    Console.Write(n);
    n++;
} while (n < 5);
// Output:
// 01234
```

Jump Statement

- Break
- Continue
- Return
- Goto

Break

```
int[] numbers = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
foreach (int number in numbers)
{
    if (number == 3)
    {
        break;
    }

    Console.Write($"{number} ");
}
Console.WriteLine();
Console.WriteLine("End of the example.");
// Output:
// 0 1 2
// End of the example.
```

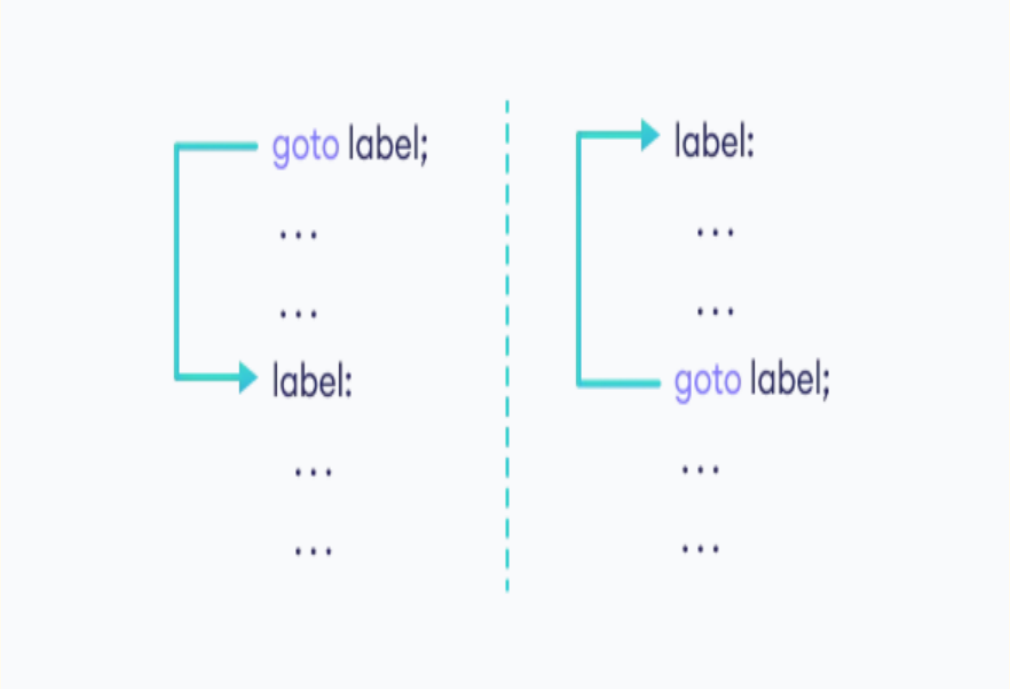
Continue

```
for (int i = 0; i < 5; i++)
{
    Console.Write($"Iteration {i}: ");

    if (i < 3)
    {
        Console.WriteLine("skip");
        continue;
    }

    Console.WriteLine("done");
}
// Output:
// Iteration 0: skip
// Iteration 1: skip
// Iteration 2: skip
// Iteration 3: done
// Iteration 4: done
```

Return

```
Console.WriteLine("First call:");
DisplayIfNecessary(6);

Console.WriteLine("Second call:");
DisplayIfNecessary(5);

void DisplayIfNecessary(int number)
{
    if (number % 2 == 0)
    {
        return;
    }

    Console.WriteLine(number);
}
// Output:
// First call:
// Second call:
// 5
```

Goto



Keywords

- Namespace
- Assembly
- Access Modifiers

Namespace

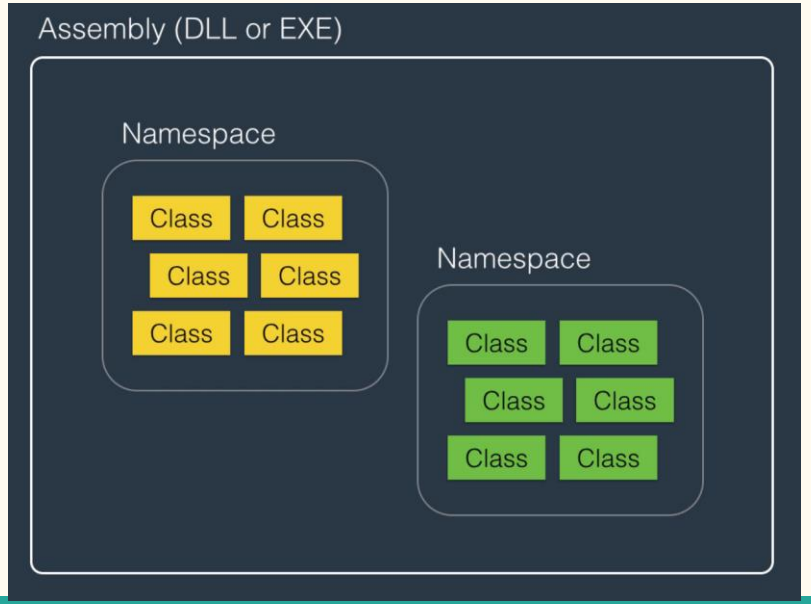
The **namespace** keyword is used to declare a scope that contains a set of related objects. You can use a namespace to organize code elements and to create globally unique types.

- You can think is as a package that includes classes that can be used elsewhere
- *namespace is often the file name

```
namespace SampleNamespace
{
    class SampleClass { }

    interface ISampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace Nested
    {
        class SampleClass2 { }
    }
}
```

Assembly

An assembly is **a collection of types and resources that are built to work together and form a logical unit of functionality**. Assemblies take the form of executable (.exe) or dynamic link library (. dll) files and they are a single unit of deployment of .NET applications.



Access Modifier

Access modifiers are keywords used to specify the declared accessibility of a field, method, constructor & class

Those access modifier allows you to grant and prevent access:

- public
- protected
- internal
- private
- protected-internal
- privare-protected

Access Modifier

Caller's location	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`
Within the class	✓	✓	✓	✓	✓	✓
Derived class (same assembly)	✓	✓	✓	✓	✓	✗
Non-derived class (same assembly)	✓	✓	✗	✓	✗	✗
Derived class (different assembly)	✓	✓	✓	✗	✗	✗
Non-derived class (different assembly)	✓	✗	✗	✗	✗	✗

https://code-maze.com/csharp-access-modifiers/

Recap

- Comment
- Variable and Data Type
  - int, long, float, double, enum, nullable, string
- Operator
- Flow Control
  - for, foreach, while
- Keywords
  - namespace, access modifier

Question?

.NET Full Stack Development Program

—

Day 3 OOP

Outline

- Class and Object
- Object-oriented programming (OOP)
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstract
- Object class

OOP Intro

- OOP stands for Object-Oriented Programming
- The goal of OOP is to group up some data and operations as a single unit called an “Object”
- Object is a small unit in the program that represents a real-world entity

Class

- C# is an object-oriented programming language, everything in C# is associated with classes and objects.
- A class is a “blueprint” for creating objects.

Class

- To create a class we need to use the keyword – ***class***
- Normally we will create a class within a namespace

```
class Employee // Please use Pascal Case to name your class
{
  // states - member fields

  // behaviors - member methods

  // constructors
}
```

Class Members

- Fields and methods inside classes are often referred to as “Class Members”.
- The variables inside a class are called fields.
- Methods are used to perform certain actions.

```
4 references
public enum Gender
{
    Man,
    Woman
}
9 references
public class Employee
{
    //Fields - to describe the state of Employee
    public string _name;
    public int _age;
    public Gender _gender;
    public Employee _manager;

    //Methods - to describe the behavior of Employee
    1 reference
    public void SayHello()
    {
        Console.WriteLine("Hello");
    }


}
```

Constructors

- A constructor is a **special method** that is used to initialize objects.
- It can be used to set initial values for fields

```
public class Employee
{
    //Fields - to describe the state of Employee
    public string _name;
    public int _age;
    public Gender _gender;
    public Employee _manager;

    //default constructor
    1 reference
    public Employee() { }

    //a constructor that can initialize name, age and gender fields
    2 references
    public Employee(string name, int age, Gender gender)
    {
        _name = name;
        _age = age;
        _gender = gender;
    }

    //Methods - to describe the behavior of Employee
    1 reference
    public void SayHello()
    {
        Console.WriteLine("Hello");
    }
}
```

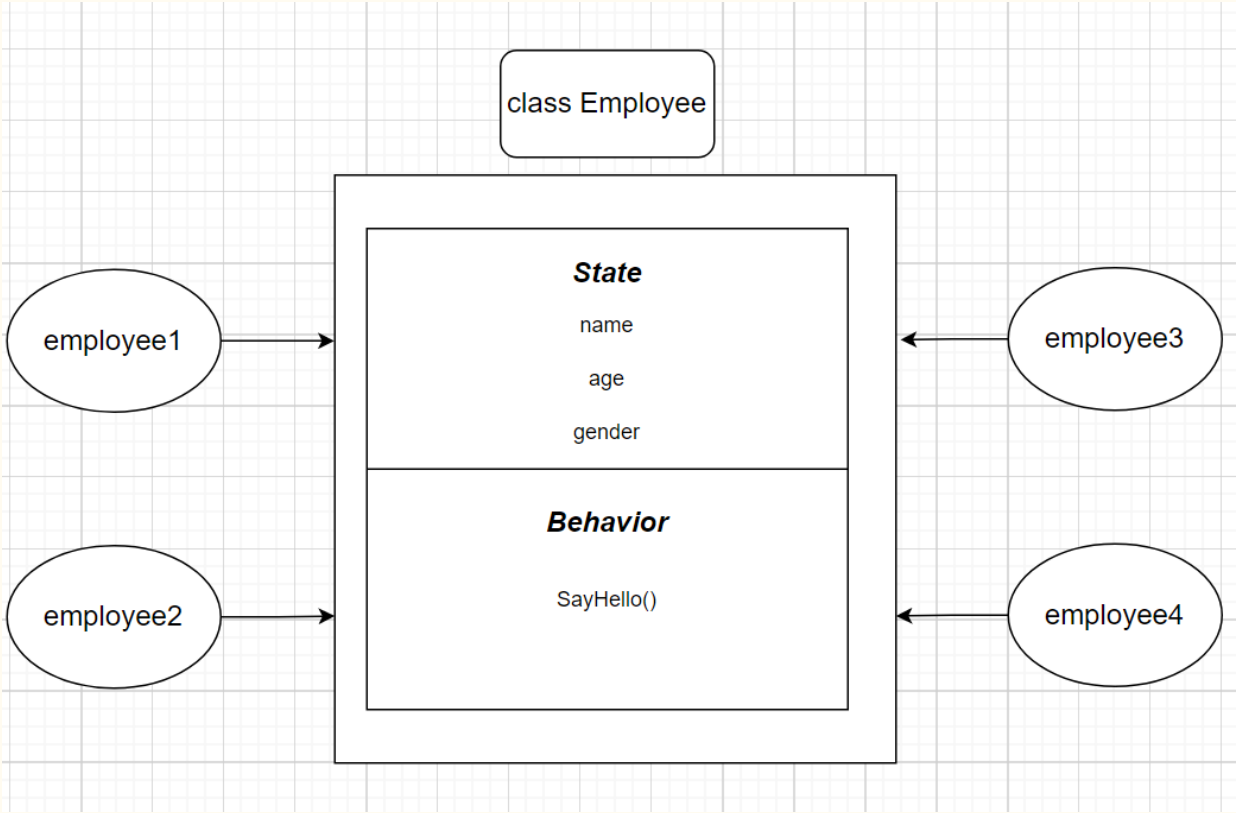
Object

- An **object** is the instance of the class, it has states and behaviors as well.
- The state of an object is stored in fields (variables), while methods (functions) display the object's behavior

- To create an object of a class, specify the class name. followed by the object name, and use the keyword new.

```
static void Main(string[] args)
{
    //use default constructor to instanciate an object called rose
    Employee rose = new Employee();
    rose._name = "Rose";
    rose._gender = Gender.Woman;
    rose._age = 22;
    rose.SayHello();
    //use the parameterized constructor to instanciate an object of Employee class called jack
    Employee jack = new Employee("Jack",23,Gender.Man);
    Console.WriteLine("Employee name: {0}, age: {1}, gender: {2}", jack._name, jack._age, jack._gender);
```

Object vs Class



Static

- declare a static member, which belongs to the **class** itself rather than to a specific object
  - Static
    - Class
    - Compile time
  - Non-static
    - Object
    - Runtime

Static Variable

- Static variable
  - Static fields are stored outside of the object
  - Static fields are common to all objects of a class

Class variable vs. Instance variable

	Class variable	Instance variable
Declaration	declared with ***static*** keyword	declared without static keyword
Storage	stored in **class's memory**	stored in the respective object
Representation	represents **common data** to all objects of the class	represents data related to the respective object
Accessibility	can be accessed by **class only**	can be accessed by objects

Static Method

- Static method
  - static method vs. instance method

	static Method	Non-*static* Method
Access instance variables?	no	yes
Access *static* class variables?	yes	yes
Call *static* class methods?	yes	yes
Call non-*static* instance methods?	no	yes
Use the object reference *this*?	no	yes

Static Class

- Static class
  - If the static keyword is applied to a class, all the members of the class must be static

This keyword

This keyword refers to the current object;

Usage:

- ***this*** is often used to differentiate between the constructor parameters and class fields if they both have the same name.

- ***this*** is also used as a modifier of the first parameter of an extension method.

```
class Employee
{
    //Fields - to describe the state of Employee
    public string name;
    public int age;
    public Gender gender;
    public Employee manager;

    //default constructor
    1 reference
    public Employee() { }

    //a constructor that can initialize name, age and gender fields
    1 reference
    public Employee(string name, int age, Gender gender)
    {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }
```

Extension Method

- It is a new feature that has been added in C# 3.0 which allows us to add new methods into a non-static class without editing the code of the class.

- Extension methods must be defined only under the static class.

- Syntax:

```
[access_modifier] static [return_type] [function_name]
(this [extension_class_name] [binding param], [param list])
{
//your extension methods here
}
```

Extension Method

```
7 references
class Employee
{
    //Fields - to describe the state of Employee
    public string name;
    public int age;
    public Gender gender;
    public Employee manager;

    //default constructor
    1 reference
    public Employee() { }

    //a constructor that can initialize name, age and gender fields
    1 reference
    public Employee(string name, int age, Gender gender)
    {
        this.name = name;
        this.age = age;
        this.gender = gender;
    }


    //Methods - to describe the behavior of Employee
    1 reference
    public void SayHello()
    {
        Console.WriteLine("Hello");
    }


}
```

```
//Extension class
0 references
static class ExtensionForEmployee
{
    1 reference
    public static void OpenTheDoor(this Employee value)
    {
        Console.WriteLine("Door Opened...");
    }


    1 reference
    public static void MorningGreeting(this Employee value, string empName)
    {
        Console.WriteLine($"Good Morning, {empName}!");
    }
}
```

```
//Extension Method
Employee han = new Employee("Han", 32, Gender.Man);
han.OpenTheDoor();
han.MorningGreeting("Mark");
```

Object-Oriented Programming

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

Encapsulation

- Encapsulation is hiding information.

- How
  - Declare fields/variables as private.
  - Provide public get and set methods, to access and update the value of a private field.

- Why
  - Flexibility — Internal logic changes won’t affect the caller of the method
  - Reusability — Encapsulated code can be used by different callers
  - Maintainability — Operations on encapsulated unit won’t affect other parts

Property

- Property is an example of Encapsulation, we can use property to access and update the value of a private field.

```
0 references
internal class People
{
    private string name; // this is a private field

    0 references
    public string Name // this is a public property
    {
        get { return name; } // get method to return the people name
        set { name = value; }// set method to change the people name to an assigned value
    }
}
```

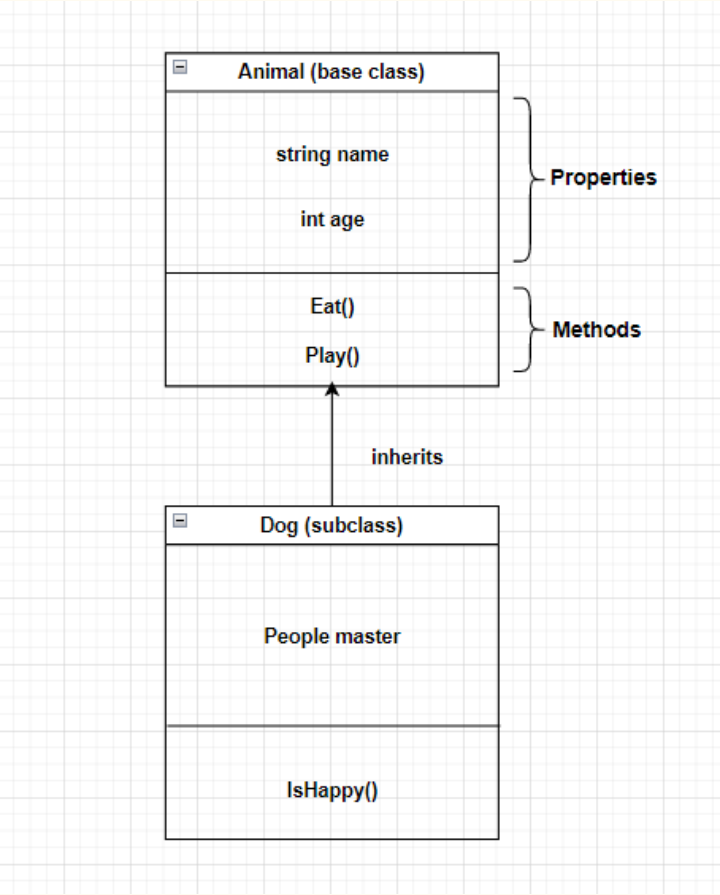
Automatic Property

```
2 references
internal class People
{
    //automatic property
    2 references
    public string Name { get; set; }

    // we can also make the property to get only or set only to meet our needs
    2 references
    public int WorkingHoursPerDay { get; } = 8;
```

```
//Property Example
People tommy = new People();
tommy.Name = "Tommy"; // use property to set the name of People
Console.WriteLine(tommy.Name); // use property to get the name of People
tommy.WorkingHoursPerDay = 15; // this property is read only, we cannot modify it
Console.WriteLine(tommy.WorkingHoursPerDay);
```

Inheritance

- Is the ability to derive something specific from something generic.
- A class can inherit the features of another class and add its own modification.
- The parent class is the base class and the child class is known as the subclass or derived class.
- A subclass inherits all the properties and methods of the base class.
- Aids in the reuse of code.



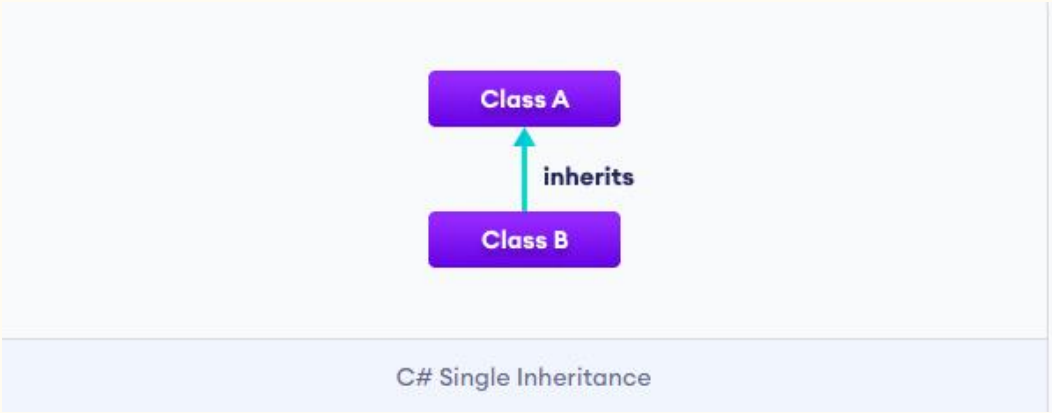
Inheritance

- Syntax
  - ***class* SubClass : BaseClass**
  - It declares that SubClass inherits the base class BaseClass
- IS-A relationship
  - Inheritance implies that there is an IS-A relationship between subclass and base class
  - Eg. Dog is an Animal; Car is a Vehicle; Triangle is a Shape

Type of Inheritance

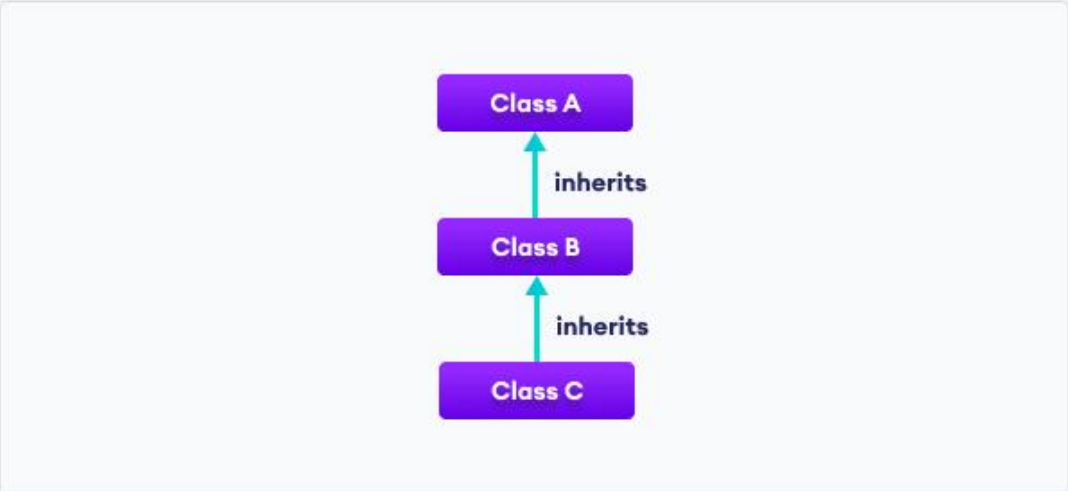
- Single
- Multiple
- Multilevel
- Hierarchical
- Hybrid

Single Inheritance



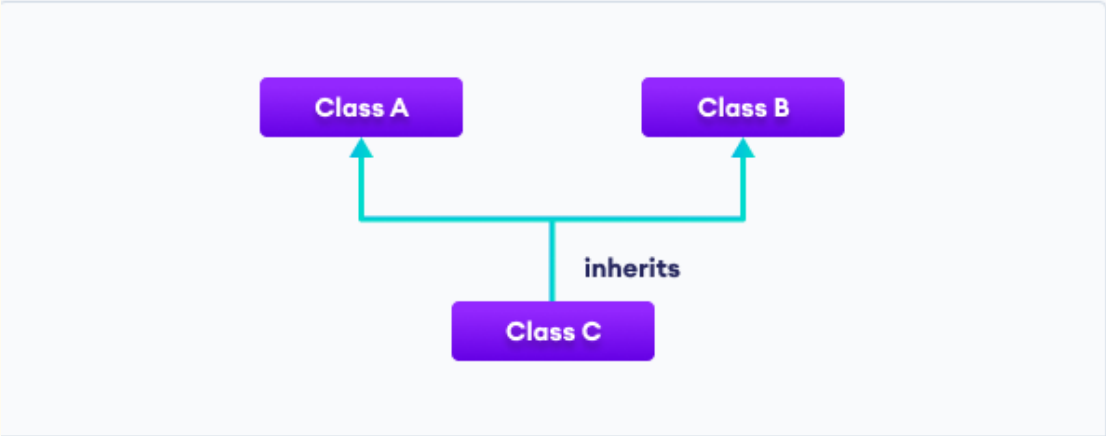
C# Single Inheritance

Multilevel Inheritance

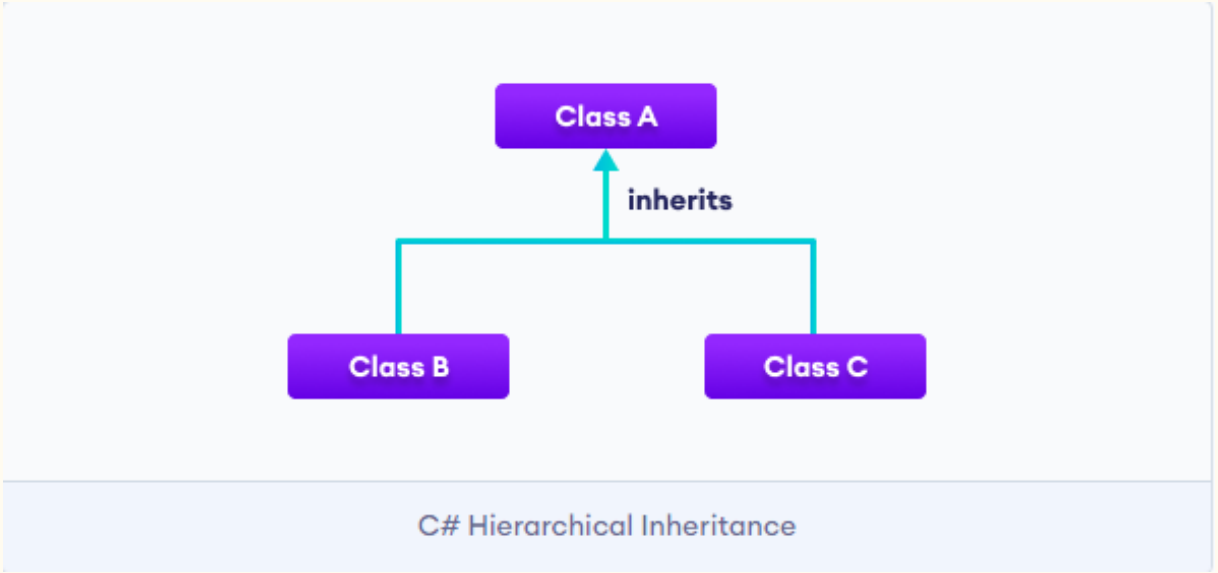


C# Multilevel Inheritance

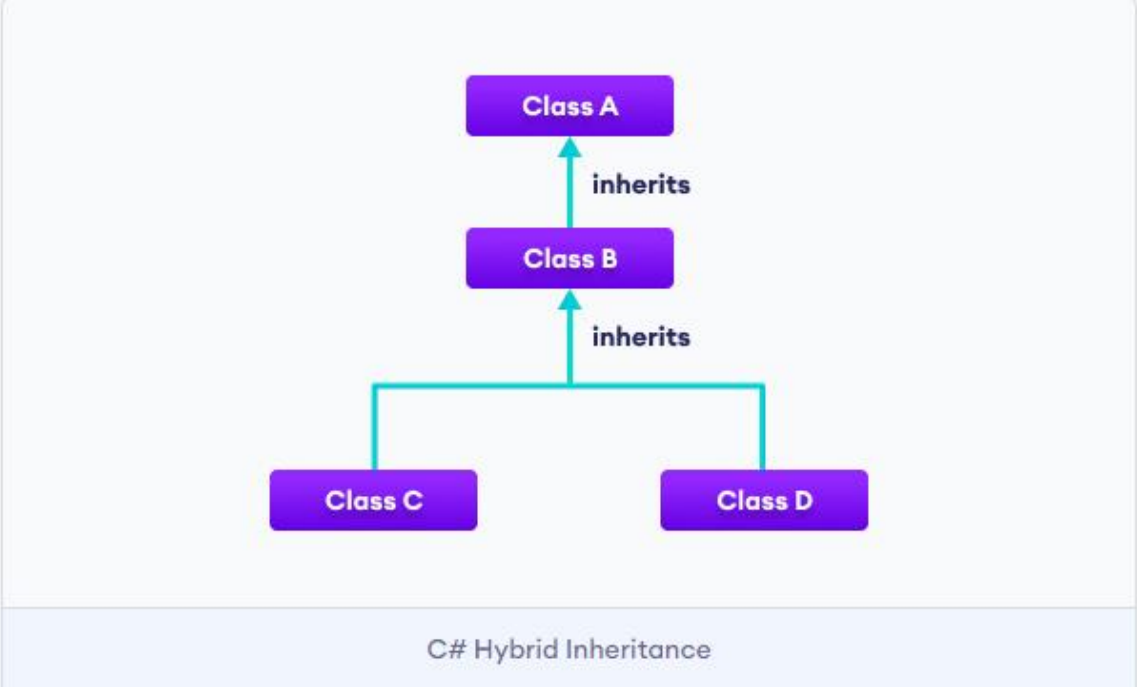
Multiple Inheritance



Hierarchical Inheritance



Hybrid Inheritance



C# Hybrid Inheritance

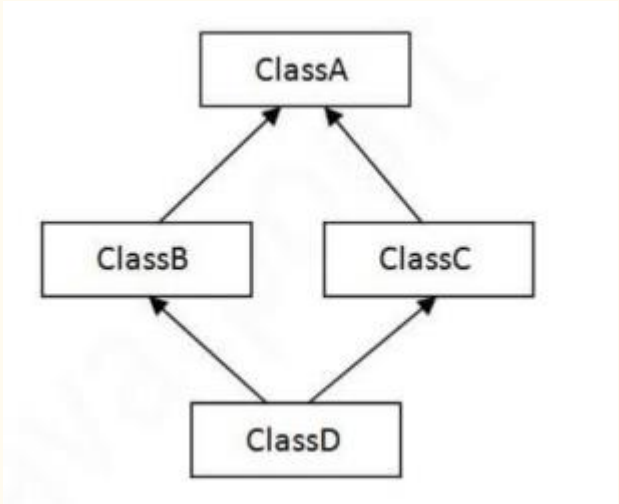
Inheritance in C#

- C# supports the following inheritance:
  - Single
  - Multi-level
  - Hierarchical

- How about Multiple and Hybrid?

Diamond Problem

- An ambiguity that can arise as a consequence of allowing multiple inheritance



Is there any way to resolve the ambiguity caused by the diamond problem?

Interface

- Like a class, an interface can have methods, but the methods declared in the interface are by default abstract (only method signature, no method body)

    - Interfaces specify what a class or a struct must do rather than how to do.
    - It defines a contract, any class or struct that implements that contract must provide an implementation of the members defined in the interface.
    - If a class implements an interface and does not provide method bodies for all functions specified in the interface, then the class must be declared abstract.

Interface

- Syntax
  - public *interface* IShape {}
  - public class Triangle **:** IShape {}

- C# allows **multiple** inheritance of interface

```
1 reference
interface IShape
{
    1 reference
    void PrintShape(); // abstract by default
}
0 references
class Triangle : IShape
{
    //must implement the abstract method in IShape interface
    1 reference
    public void PrintShape()
    {
        Console.WriteLine("Triangle");
    }
}
```

```
interface ISpeak
{
    2 references
    void SayHello(); // abstract by default

}
2 references
interface ISay
{
    2 references
    void SayHello(); //abstract by default

}
2 references
class Person : ISpeak, ISay
{
    4 references
    public void SayHello()
    {
        Console.WriteLine("Hello") ;
    }
}
```

Aggregation

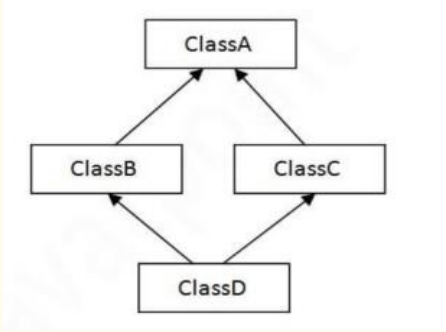
- If a class has an entity reference, it is known as Aggregation
- Aggregation represents HAS-A relationship

```
internal class Employee
{
    0 references
    public int Id { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public Address Address { get; set; } // Employee Has-An Address


}

1 reference
internal class Address
{
    0 references
    public string Street { get; set; }
    0 references
    public string City { get; set; }
    0 references
    public int ZipCode { get; set; }
}
```

Aggregation

- Code reuse is also best achieved by aggregation when there is no is-a relationship.
- If we don’t have the Address class
  - Have to add all street, city information in employee
  - If we introduce an object called Company later which has the address too, we will have to add same attributes to Company object again

Aggregation



- How does Aggregation solve the diamond problem?
  - Instead of inheriting class B and C, introducing the B and C as the instance variables in class D
  - Thus, we can use b.DoSomething() or c.DoSomething() to eliminate the ambiguity

Inheritance vs. Aggregation

- Inheritance should be used only if the relationship Is-A is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

Polymorphism

- **Polymorphism** in c# is a concept by which we can perform a single action in different ways.
  - The word "poly" means many and "morphs" means forms. So polymorphism means many forms.
- There are two types of polymorphism in c#
  - **Compile time** polymorphism (achieved by **method overloading**)
  - **Runtime** polymorphism (achieved by **method overriding**)

Compile time vs. Runtime

- Compile time — the instance where the code you entered is converted to executable

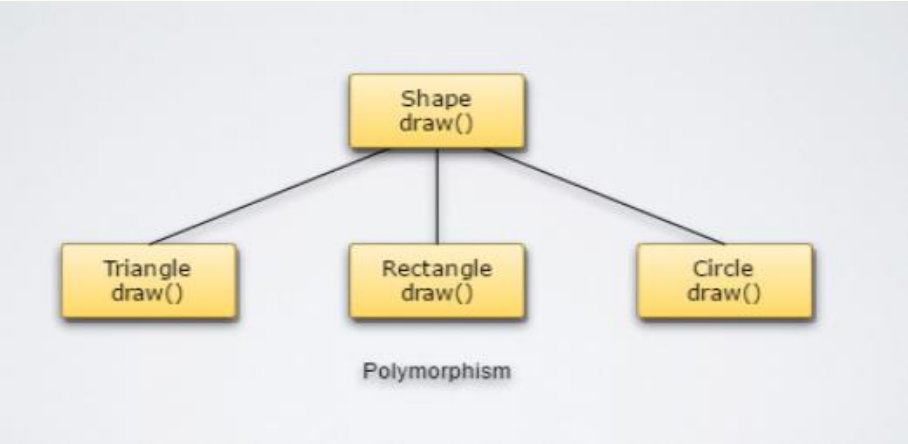
- Runtime — the instance where the executable is running

Runtime Polymorphism

- The form is determined at runtime
- Method **overriding**
  - a method in a subclass has the **same name, return type, and parameters** as a method in its base class, then the method in the subclass is said to override the method in the base class
  - When an overridden method is called through the subclass object, it will always refer to the version of the method defined by the subclass. The base class version of the method is hidden.

Virtual and Override Keywords

- The ***virtual*** keyword is used to specify the virtual method in the base class, and the method with the same signature that needs to be overridden in the derived class is preceded by ***override*** keyword.
    - By default, methods are **non-virtual**. You cannot override a non-virtual method.
    - You cannot use the ***virtual*** keyword with the ***static***, ***private***, ***abstract***, or ***override*** keywords




```
//base class
3 references
public class Shape
{
    //virtual method
    3 references
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks!");
    }
}
```

```
public class Circle : Shape
{
    //override the virtual method in base class
    1 reference
    public override void Draw()
    {
        Console.WriteLine("Drawing a circle");
    }
}
0 references
public class Rectangle : Shape
{
    //override the virtual method in base class
    1 reference
    public override void Draw()
    {
        Console.WriteLine("Drawing a rectangle");
    }
}
0 references
public class Triangle : Shape
{
    //override the virtual method in base class
    1 reference
    public override void Draw()
    {
        Console.WriteLine("Drawing a triangle");
    }
}
```

Compile Time Polymorphism

- The form is determined at compile time
- Method **overloading**
  - a class has multiple methods that have the same name but different parameters
    - Different number of parameters
    - Different data types of parameters
  - Method overloading increases the readability of the program

Compile Time Polymorphism

```
public class Program
{
    0 references
    public static void PrintArea(int x, int y)
    {
        Console.WriteLine(x * y);
    }
    0 references
    public static void PrintArea(int x)
    {
        Console.WriteLine(x * x);
    }
    0 references
    public static void PrintArea(int x, double y)
    {
        Console.WriteLine(x * y);
    }
    0 references
    public static void PrintArea(double x)
    {
        Console.WriteLine(x * x);
    }
```

Overriding vs. Overloading

Overriding vs Overloading in C#	
Overriding in C# is to provide a specific implementation in a derived class method for a method already existing in the base class.	Overloading in C# is to create multiple methods with the same name with different implementations.
Parameters	
In C# Overriding, the methods have the same name, same parameter types and a same number of parameters.	In C# Overloading, the methods have the same name but a different number of parameters or a different type of parameters.
Occurrence	
In C#, overriding occurs within the base class and the derived class.	In C#, overloading occurs within the same class.
Binding Time	
The binding of the overridden method call to its definition happens at runtime.	The binding of the overloaded method call to its definition happens at compile time.
Synonyms	
Overriding is called as **runtime polymorphism**, **dynamic polymorphism** or **late binding**.	Overloading is called as **compile time polymorphism**, **static polymorphism** or **early binding**.

If I want to use the implementation in the base class, how to refer to the base class?

Base

- The **base** keyword is used to access members of the base class from within a derived class
- Usage
  - Call a method on the base class that has been overridden by another method.
  - Use base() to specify which base-class constructor should be called when creating instances of the derived class.

C#

```
public class Person
{
    protected string ssn = "444-55-6666";
    protected string name = "John L. Malgraine";

    public virtual void GetInfo()
    {
        Console.WriteLine("Name: {0}", name);
        Console.WriteLine("SSN: {0}", ssn);
    }
}
class Employee : Person
{
    public string id = "ABC567EFG";
    public override void GetInfo()
    {
        // Calling the base class GetInfo method:
        base.GetInfo();
        Console.WriteLine("Employee ID: {0}", id);
    }
}

class TestClass
{
    static void Main()
    {
        Employee E = new Employee();
        E.GetInfo();
    }
}
/*
Output
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
*/
```

C#

```
public class BaseClass
{
    int num;

    public BaseClass()
    {
        Console.WriteLine("in BaseClass()");
    }

    public BaseClass(int i)
    {
        num = i;
        Console.WriteLine("in BaseClass(int i)");
    }

    public int GetNum()
    {
        return num;
    }
}

public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base()
    {
    }

    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i)
    {
    }

    static void Main()
    {
        DerivedClass md = new DerivedClass();
        DerivedClass md1 = new DerivedClass(1);
    }
}
/*
Output:
in BaseClass()
in BaseClass(int i)
*/
```

How can we prevent one method from being overridden?

Sealed

- Method – prevent overriding specific virtual methods

```
class X
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}

class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2"); }
}

class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("Z.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

- Class – prevent inheritance
  - sealed class SealedClass {}

Abstraction

- **Hiding internal details** and showing functionality
  - eg. phone call, we don't know the internal processing
- Abstraction is achieved by creating either **Abstract Classes** or **Interfaces** on top of your class
- Why
  - Hide the unnecessary things from user so providing easiness.
  - Hiding the internal implementation of software so providing security

```
//To declare an abstract class
1 reference
abstract class Animal
{
    //Abstract method
    2 references
    public abstract void AnimalSound();

    // normal method
    1 reference
    public void Sleep()
    {
        Console.WriteLine("Zzz");
    }
}

// subclasses should override the abstract methods
2 references
class Dog : Animal
{
    2 references
    public override void AnimalSound()
    {
        Console.WriteLine("Woof, Woof"); ;
    }
}
```

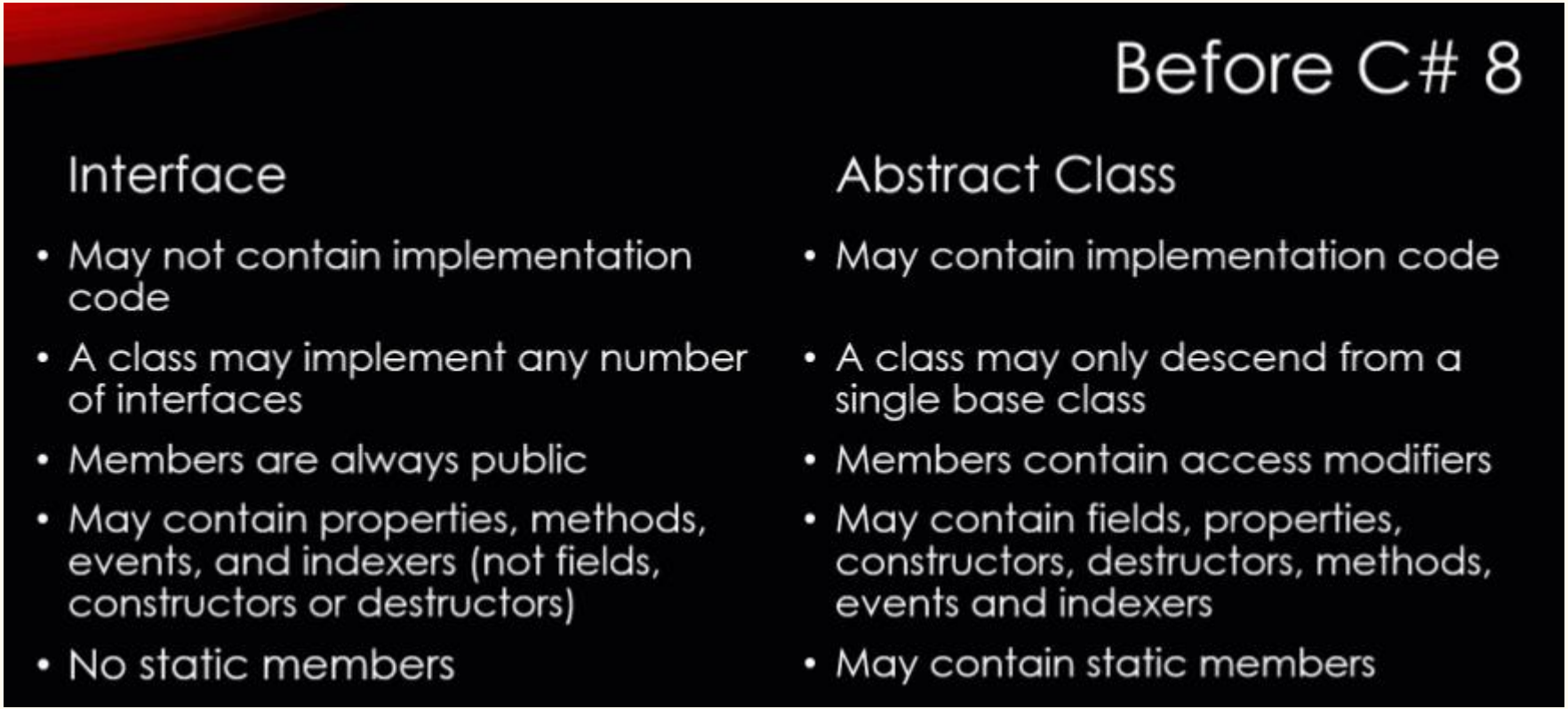
Abstract Class

- Abstract classes are classes with a generic concept, not related to a specific class.
- Abstract classes define **partial behavior** and leave the rest for the subclasses to provide
- Contain zero or more abstract methods.
- Abstract method contains no implementation (like the method in Interface)
- Abstract classes **cannot be instantiated,** but they can have a reference variable
- If the subclasses do not override the abstract methods of the abstract class, then it is mandatory for the subclasses to tag themselves as abstract.

Why Abstract Class

- To force the same name and signature pattern in all the subclasses
- To have the flexibility to code these methods with their own specific requirements
- To prevent accidental initialization
- To define common attributes or methods

Abstract Class vs. Interface



Abstract Class vs. Interface

After C# 8

Interface

- ~~May not contain implementation code~~
- A class may implement any number of interfaces
- ~~Members are always public~~
- ~~May contain properties, methods, events, and indexers (not fields, constructors or destructors)~~
- ~~No static members~~

Abstract Class

- May contain implementation code
- A class may only descend from a single base class
- Members contain access modifiers
- May contain fields, properties, constructors, destructors, methods, events and indexers
- May contain static members

Object Class

Object Class in C#

All types in the .NET type system **implicitly inherit** from **Object** or a type derived from it. The common functionality of **Object** is available to **any type**.

- The Object class is beneficial if you want to refer any object whose type you don't know. Notice that the parent class reference variable can refer to the child class object, known as **upcasting**

Methods of Object Class

Equals(Object)	Determines whether the specified object is equal to the current object.
Equals(Object, Object)	Determines whether the specified object instances are considered equal.
Finalize()	Allows an object to try to free resources and perform other cleanup operations before it is reclaimed by garbage collection.
GetHashCode()	Serves as the default hash function.
GetType()	Gets the Type of the current instance.
MemberwiseClone()	Creates a shallow copy of the current Object.
ReferenceEquals(Object, Object)	Determines whether the specified Object instances are the same instance.
ToString()	Returns a string that represents the current object.

Object Casting

```
2 references
class Grandparent
{
    0 references
    public string Name { get; set; } = "Grandparent";
}

6 references
class Parent : Grandparent
{
    0 references
    public bool IsHealthy { get; set; } = true;
    0 references
    public List<Child> Children { get; set;} = new List<Child>();
}

5 references
class Child : Parent
{
    0 references
    public Parent TheParent { get; set; }
}
```

```
0 references
static void Main(string[] args)
{
    Grandparent grandparent = new Parent(); //up cast
    Parent parent1 = grandparent;
    Parent parent2 = (Parent)grandparent; // down cast
    Child child1 = grandparent;
    Child child2 = (Child)grandparent; // down cast


}
```

Recap

- Class and Object
- Object-oriented programming (OOP)
- Encapsulation
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction
- Object class

Question?

.NET Full Stack Development Program

—

Day 4 Collections

Outline

- Array
- Collections
  - Generic Collections
    - List, LinkedList, Dictionary, HashSet, SortedList, Stack, Queue
  - Non-generic Collections
    - ArrayList, HashTable, SortedList, Stack, Queue
- Comparisons and Sorts within Collections

Collection

- Group of objects
- It is not specified whether they are
  - Ordered / not ordered
  - Duplicated / not duplicated
- Following constructors are common to all classes implementing Collection
  - T()
  - T(){...}
  - T(Collection c)

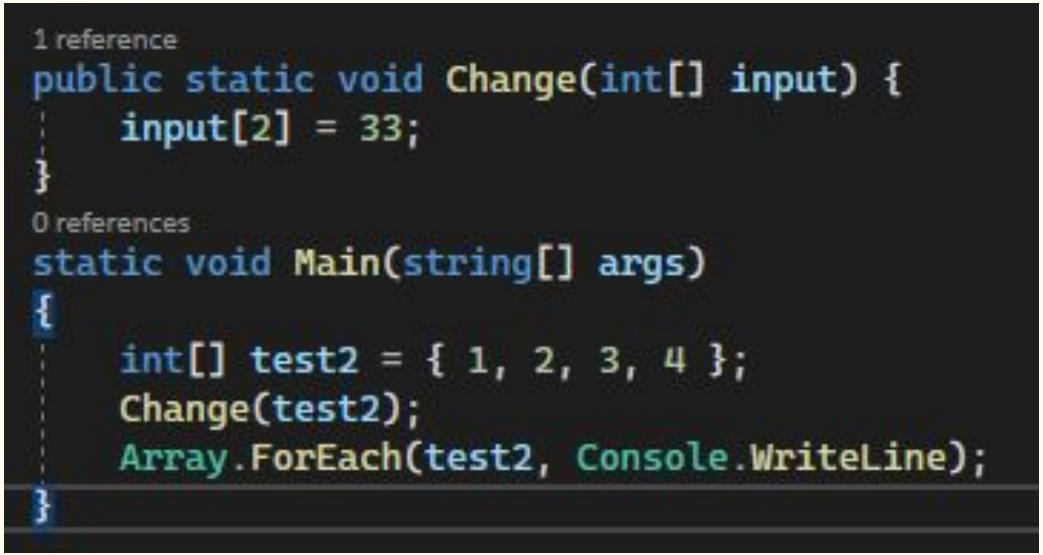
Array

- Can store multiple variables of the **same type**
- Size of the array is **fixed** when the array instance is created
- If you want the array to store elements of any type, you can specify **object** as its type
- The default values of numeric array elements are set to **zero**, and reference elements are set to **null**

```
// Declare a single-dimensional array of 5 integers.
int[] array1 = new int[5];

// Declare and set array element values.
int[] array2 = new int[] { 1, 3, 5, 7, 9 };

// Alternative syntax.
int[] array3 = { 1, 2, 3, 4, 5, 6 };
```

Array

- Passing Arrays as Argument



Collections

- Generic Collections
  - Generic Collections work on the **specific type** that is specified in the program whereas non-generic collections work on the **object** type.
  - Using System.Collections.Generic;
  - List, LinkedList, Dictionary, HashSet, SortedList, Stack, Queue
- Non-generic Collections
  - In non-generic collections, each element can represent a value of a **different type**. The collection size is not fixed. Items from the collection can be added or removed at runtime
  - Using System.Collections;
  - ArrayList, HashTable, SortedList, Stack, Queue

Generic Collection

- List
  - List class is a collection that can be used for **specific types.**
  - List is a class that is similar to an array, but the size is not fixed
  - Elements can be added / removed at runtime.
  - Ex. `List<int> al = new List<int>();`

```
// Create a list of strings by using a
// collection initializer.
var salmons = new List<string> { "chinook", "coho", "pink", "sockeye" };

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye
```

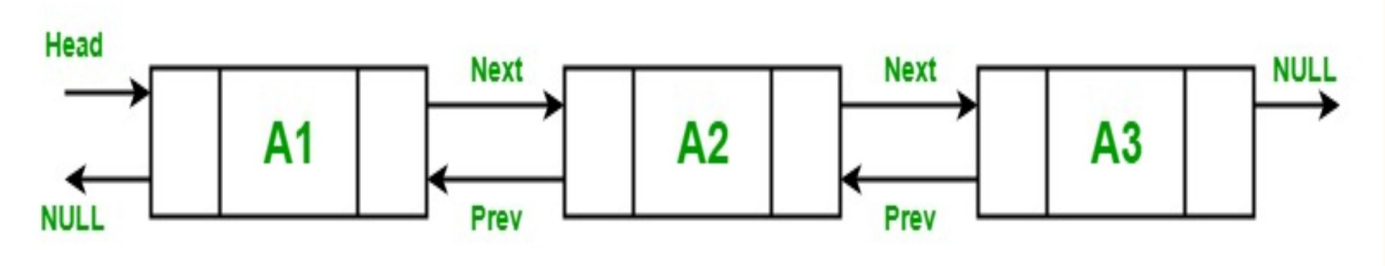
Generic Collection

- List
  - access
  - Remove()
  - RemoveAt()
  - Contains()
  - IndexOf()
  - LastIndexOf()

Generic Collection

- LinkedList
  - a general-purpose linked list (doubly linked)
  - LinkedList<T> provides separate nodes of type **LinkedListNode<T>**, so insertion and removal are **O(1)** operations.

```
// Create the link list.
string[] words =
    { "the", "fox", "jumps", "over", "the", "dog" };
LinkedList<string> sentence = new LinkedList<string>(words);
sentence.AddFirst("today");
// Move the first node to be the last node.
LinkedListNode<string> mark1 = sentence.First;
sentence.RemoveFirst();
sentence.AddLast(mark1);
LinkedListNode<string> current = sentence.FindLast("the");
sentence.AddAfter(current, "old");
sentence.AddAfter(current, "lazy");
```

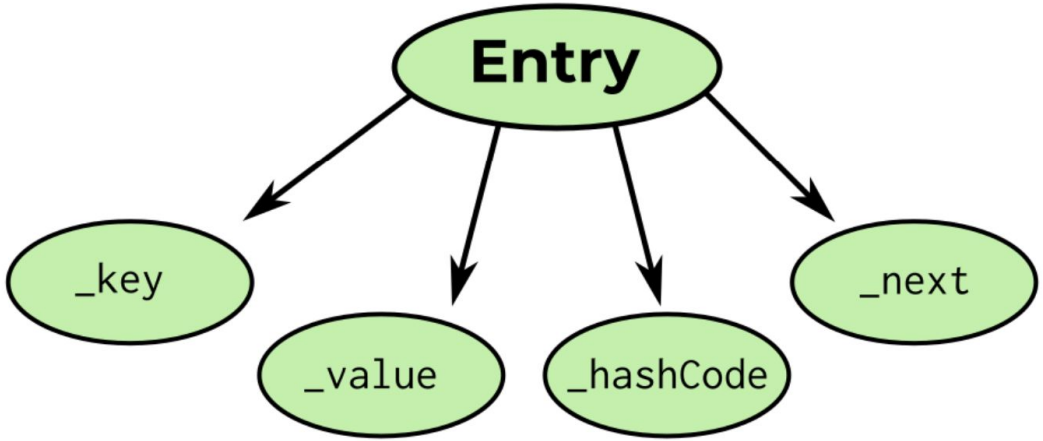

Head
A1
Next
A2
Next
A3
NULL
NULL
Prev
Prev

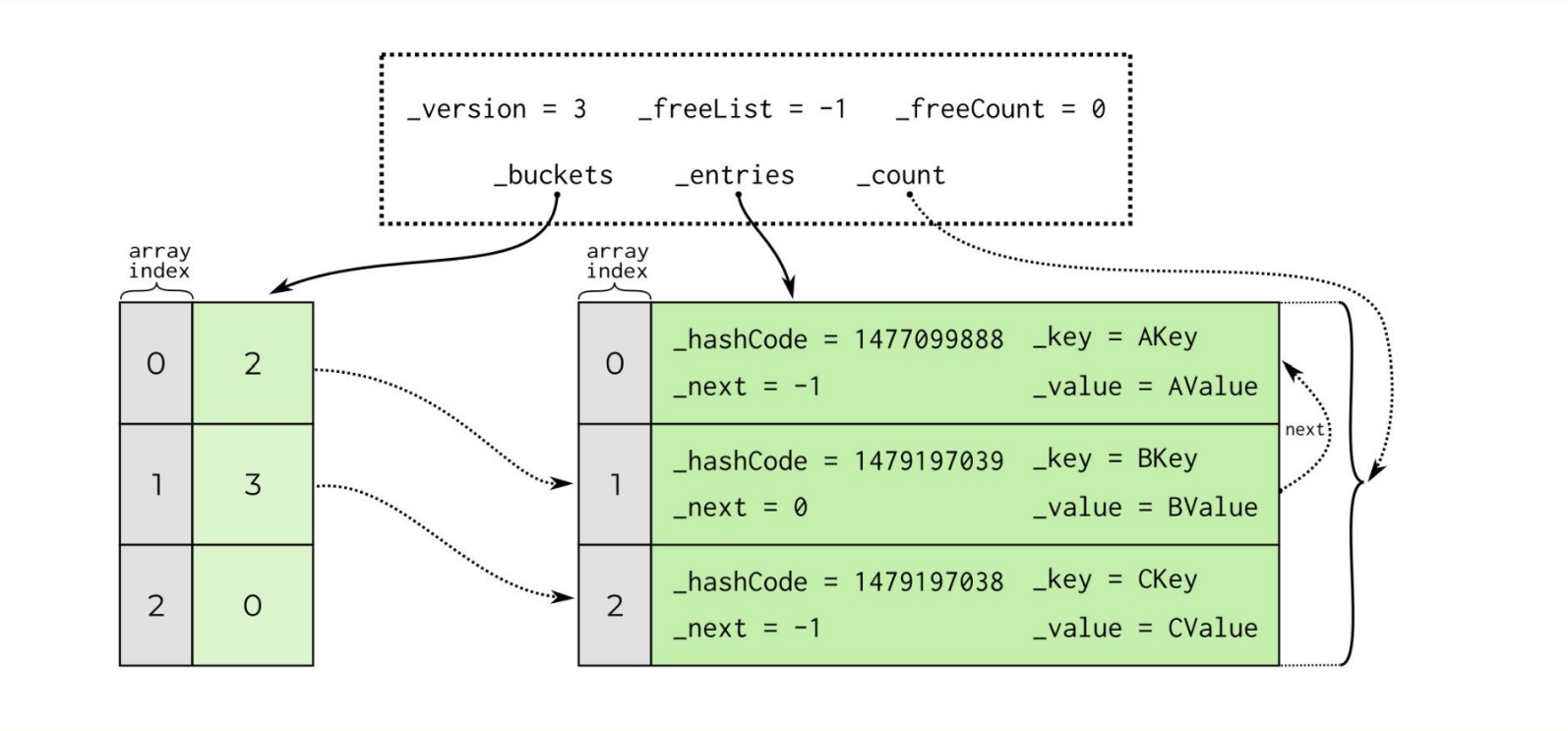
Generic Collection

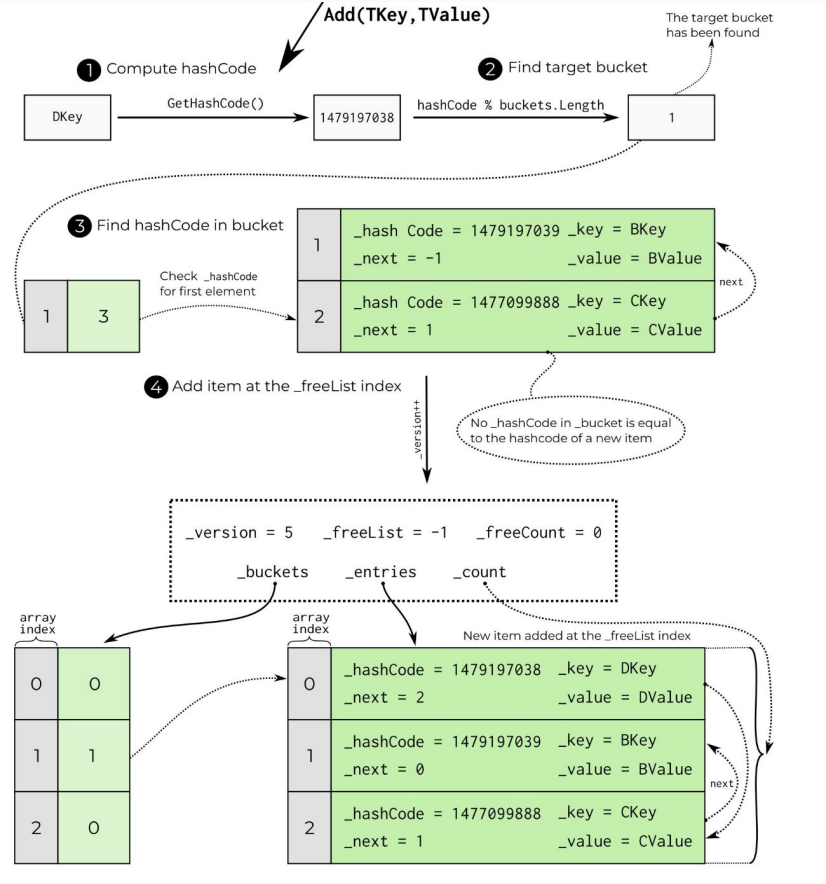
- Dictionary
  - represents the items as a combination of a key and value
  - access the value based on the key

```
Dictionary<int, string> dct = new Dictionary<int, string>();
dct.Add(1, "cs.net");
dct.Add(2, "vb.net");
dct.Add(3, "vb.net");
dct.Add(4, "vb.net");
foreach (KeyValuePair<int, string> kvp in dct)
{
    Console.WriteLine(kvp.Key + " " + kvp.Value);
}
```

Entry struct




_version = 3
_freeList = -1
_freeCount = 0
_buckets
_entries
_count
array index
0
2
1
3
2
0
array index
0
_hashCode = 1477099888
_key = AKey
_next = -1
_value = AValue
1
_hashCode = 1479197039
_key = BKey
_next = 0
_value = BValue
2
_hashCode = 1479197038
_key = CKey
_next = -1
_value = CValue
next


Add(TKey,TValue)
1 Compute hashCode
DKey
GetHashCode()
1479197038
2 Find target bucket
hashCode % buckets.Length
1
The target bucket has been found
3 Find hashCode in bucket
1
3
Check _hashCode for first element
1
_hash Code = 1479197039 _key = BKey
_next = -1 _value = BValue
2
_hash Code = 1477099888 _key = CKey
_next = 1 _value = CValue
next
4 Add item at the _freeList index
_version++
No _hashCode in _bucket is equal to the hashcode of a new item
_version = 5 _freeList = -1 _freeCount = 0
_buckets _entries _count
array index
0 0
1 1
2 0
array index
New item added at the _freeList index
0
_hashCode = 1479197038 _key = DKey
_next = 2 _value = DValue
1
_hashCode = 1479197039 _key = BKey
_next = 0 _value = BValue
2
_hashCode = 1477099888 _key = CKey
_next = 1 _value = CValue
next

Generic Collection

- Dictionary
  - Quick initialization

```
Dictionary<string, string> dict = new Dictionary<string, string> {
                                                    {"A", "Apple"},
                                                    {"B", "Banana"},
                                                    {"C", "Celery"}};
```

  - Methods
    - ContainsKey()
    - TryGetValue()
    - TryAdd()

Generic Collection

- HashSet
  - a collection that contains no duplicate elements, and whose elements are in no particular order

```
HashSet<string> cities = new HashSet<string>
{
    "Mumbai",
    "Vadodara",
    "Surat",
    "Ahmedabad",
    "Bharuch"
};
cities.Add("Mumbai");
cities.Add("Vadodara");
cities.Add("Surat");
cities.Add("Ahmedabad");
cities.Add("Bharuch");
```

Generic Collection

- SortedList
  - represents a collection of key/value pairs that are **sorted by key** based on the associated **IComparer<T>** implementation.

```
SortedList<string, string> sl = new SortedList<string, string>();
sl.Add("ora", "oracle");
sl.Add("vb", "vb.net");
sl.Add("cs", "cs.net");
sl.Add("asp", "asp.net");

foreach (KeyValuePair<string, string> kvp in sl)
{
    Console.WriteLine(kvp.Key + " " + kvp.Value);
}
```

Generic Collection

- Stack
  - It represents a last-in, first out collection of object

- Queue
  - It represents a first-in, first out collection of object

```
Stack<string> stk = new Stack<string>();
stk.Push("cs.net");
stk.Push("vb.net");
stk.Push("asp.net");
stk.Push("sqlserver");

foreach (string s in stk)
{
    Console.WriteLine(s);
}
```

```
Queue<string> q = new Queue<string>();

q.Enqueue("cs.net");
q.Enqueue("vb.net");
q.Enqueue("asp.net");
q.Enqueue("sqlserver");

foreach (string s in q)
{
    Console.WriteLine(s);
}
```

Non-generic Collection

- ArrayList
  - ArrayList class is a collection that can be used for **any types or objects.**
    - Arraylist is a class that is similar to an array, but it can be used to store values of various types.
    - An Arraylist doesn't have a specific size.
    - Any number of elements can be stored.
    - Ex. `ArrayList al = new ArrayList();`

Non-generic Collection

- HashTable
  - represents the items as a combination of a key and value

```
Hashtable ht = new Hashtable();
ht.Add("ora", "oracle");
ht.Add("vb", "vb.net");
ht.Add("cs", "cs.net");
ht.Add("asp", "asp.net");

foreach (DictionaryEntry d in ht)
{


}
```

Non-generic Collection

- SortedList
  - is a class that has the combination of arraylist and hashtable.
  - represents a collection of key/value pairs that are **sorted by key** and are accessible by key and by index

```
SortedList sl = new SortedList();
sl.Add("ora", "oracle");
sl.Add("vb", "vb.net");
sl.Add("cs", "cs.net");
sl.Add("asp", "asp.net");

foreach (DictionaryEntry d in sl)
{


}
```

Non-generic Collection

- Stack
  - It represents a last-in, first out collection of object
  - It is used when you need a last-in, first-out access of items. When you add an item in the list, it is called pushing the item and when you remove it, it is called popping the item
- Queue
  - It represents a first-in, first out collection of object
  - It is used when you need a first-in, first-out access of items. When you add an item in the list, it is called enqueue and when you remove it, it is called deque

Comparison and Sorts within Collections

Comparison (check for equality)

- If type T implements the **IEquatable<T>** generic interface, then the equality comparer is the **Equals** method of that interface.

- If type T does **not** implement **IEquatable<T>**, **Object.Equals** is used.

Default Sorting

- Array.Sort(xxx) - using the System.IComparable
- xxx.Sort() – xxx is a collection – using default comparer (System.IComparable)
  - String objects are lexicographically ordered
  - Date objects are chronologically ordered
  - Number and sub-classes are ordered numerically

Sort Order

- **IComparable<T>** interface

- **IComparer<T>** Interface

IComparable<T> interface

```
public interface IComparable
{
    int CompareTo(object? obj);
}
```

Compares the receiving object with the specified object

- Return value must be:
  - <0, if this precedes obj
  - ==0, if this has the same order as obj
  - >0, if this follows ob

```
public class Car : IComparable<Car>
{
    public string Name { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }

    public int CompareTo(Car other)
    {
        // A call to this method makes a single comparison that is
        // used for sorting.

        // Determine the relative order of the objects being compared.
        // Sort by color alphabetically, and then by speed in
        // descending order.

        // Compare the colors.
        int compare;
        compare = String.Compare(this.Color, other.Color, true);

        // If the colors are the same, compare the speeds.
        if (compare == 0)
        {
            compare = this.Speed.CompareTo(other.Speed);

            // Use descending order for speed.
            compare = -compare;
        }

        return compare;
    }
}
```

IComparer<T> Interface

```
public interface IComparer
{
    int Compare(object? x, object? y);
}
```

Compares its two arguments

- Return value must be
  - <0, if x precedes y
  - ==0, if x has the same ordering as y
  - >0, if x follows y

```
// This class is not demonstrated in the Main method
// and is provided only to show how to implement
// the interface. It is recommended to derive
// from Comparer<T> instead of implementing IComparer<T>.
public class BoxComp : IComparer<Box>
{
    // Compares by Height, Length, and Width.
    public int Compare(Box x, Box y)
    {
        if (x.Height.CompareTo(y.Height) != 0)
        {
            return x.Height.CompareTo(y.Height);
        }
        else if (x.Length.CompareTo(y.Length) != 0)
        {
            return x.Length.CompareTo(y.Length);
        }
        else if (x.Width.CompareTo(y.Width) != 0)
        {
            return x.Width.CompareTo(y.Width);
        }
        else
        {
            return 0;
        }
    }
}
```

Questions?

.NET Full Stack Development Program

—

Day 5 I/O & Exception & C#Advance

Outline

- File Handling (System.IO)
- Exception
- Delegate
- Lambda Expression

File Handling

System.IO

IO(Input and Output) is used to process the input and produce the output (read and write data). Most applications need to process some data and produce some output based on the input.

The **System.IO namespace** contains all the classes required for IO operations.

https://learn.microsoft.com/en-us/dotnet/api/system.io?view=net-7.0

File class

- C# include **static File class** to perform I/O operations on physical file.
- File class provides functionalities such as create, read/write, delete, etc. for physical files.
- Some methods in the File class:

Method	Description
static FileStream **Create**(string path)	Create or overwrite a file at the path
static bool **Exists**(string path)	Determine whether the file exists at the path
static void **Move**(string sourceFile, string destFile)	Move the source file to the destination
static void **Copy**(string sourceFile, string destFile)	Copy the source file to the destination
static void **Delete**(string path)	Delete the file at path

File class

- Read/Write methods in File class:

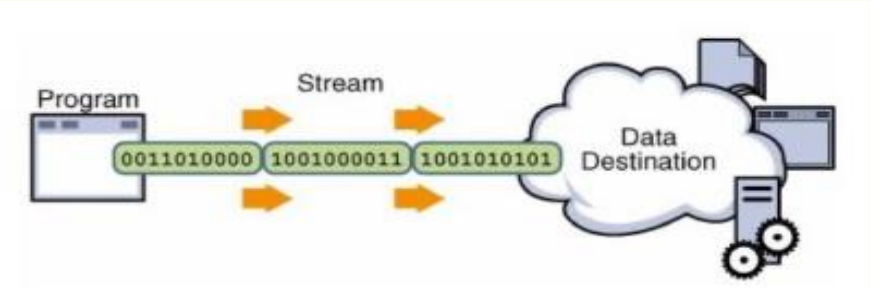
Method	Method
static void **WriteAllText**(string path, string contents)	Create or overwrite a file, write contents to the file and close the file
static string **ReadAllText**(string path)	Read from the file and return a string representaion of text

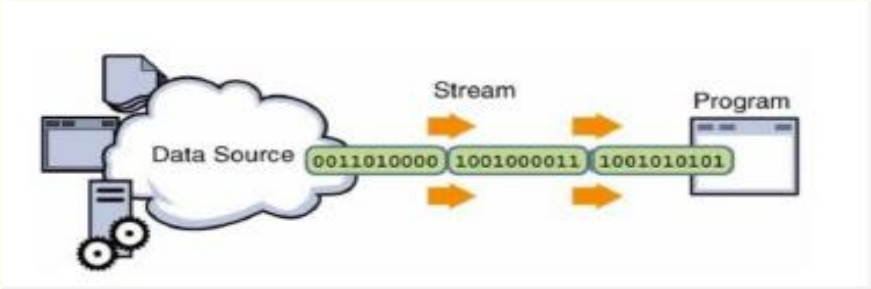
File Class

- The static File class includes various utility methods to interact with a physical file
- It’s used to perform some quick operations on physical file
- It’s **NOT** recommended to use File class for multiple operations on multiple files at the same time due to performance reasons, a non-static class would be a better choice in this situation

Stream

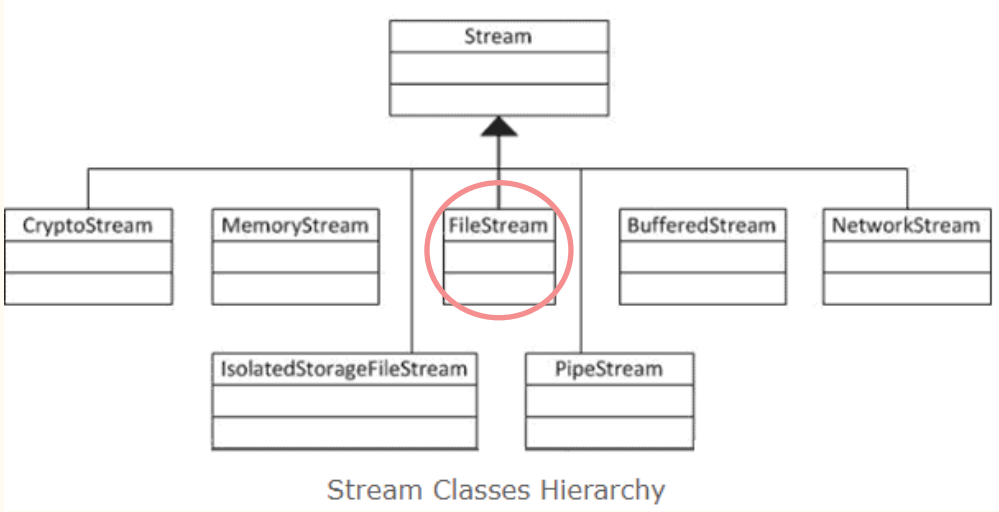
- C# uses the Stream to make IO operation fast.
- A stream is a conceptually endless flow of data. We can either read from a stream or write to a stream.
- A stream is connected to a data source or a data destination.
- It’s a good practice to close the stream after use.





Stream Class

- Stream is an abstract class that provides standard methods to transfer bytes to the source.
- The following classes inherit the Stream class to provide the functionality to Read/Write bytes from a particular source:

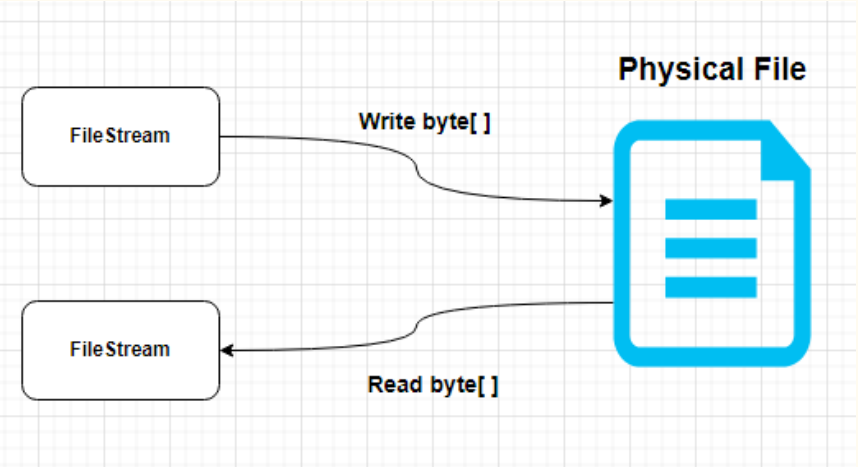


Stream Classes Hierarchy

FileStream Class

- FileStream class provides a stream for file operations. It can be used to perform both **read** and **write operations.**

- To use the FileStream we need to:
  - **Include System.IO namespace**
  - **Create an instance of the FileStream**
  - **Perform read or write operations**
  - **Close the stream**



Create an Instance of FileStream

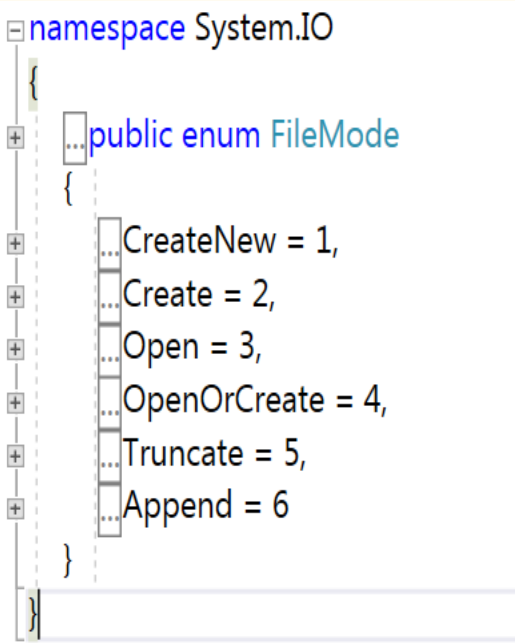
- We need an instance of the FileStream object to create new file or open an existing file.
- There are many constructors available, here are 2 simple ways:

```
//Some Constructors that are often used to create FileStream objects
public FileStream(string path, FileMode mode);
public FileStream(string path, FileMode mode, FileAccess access);
```

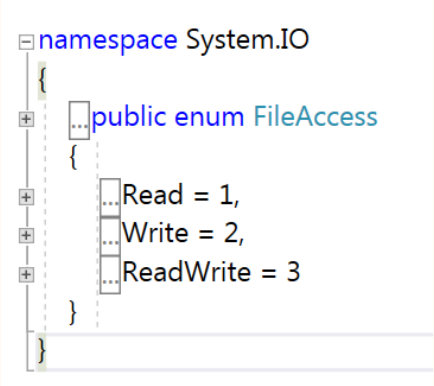
  - **Path**: the path to the file that the current FileStream object will work on;
  - **FileMode**: specifies how to deal with the file(create, open, append ...)
  - **FileAccess**: determines how the file can be accessed by the FileStream object(read, write...)

FileMode

FileMode	Description
CreateNew	It specifies that the OS should **create a new file**. If the file already exists an ***IOException*** will be thrown.
Create	It specifies that the OS should **create a new file**. If the file already exists, it will be **overwritten**.
Open	It specifies that the OS should **open an existing file**. If the file doesn't exist, a ***FileNotFoundException*** will be thrown.
OpenOrCreate	It specifies that the OS should **open an existing file**. If the file doesn't exist, a new file will be **created**. It's useful when reading from the file.
Truncate	It specifies that the OS should **open an existing file**. When the file is opened, it should be truncated so that its size is zero bytes. Attempts to read from a file opened with FileMode.Truncate causes an ***ArgumentException*** exception.
Append	It specifies that the OS should **open an existing file**, and seek to end of the file, in order to **write content at the end**. It's useful when writing to the file.



FileAccess



FileAccess	Description
Read	It gives **read access** to the file. Data can be read from the file.
Write	It gives **write access** to the file. Data can be written to the file.
ReadWrite	It gives **read and writes access** to the file. Data can be written to and read from the file.

FileStream

- Perform Read/Write operations:

```
// Read() and Write() methods provided by the FileStream
public override int Read(byte[] array, int offset, int count);
public override void Write(byte[] array, int offset, int count);
```

- Close the Stream:
    - We can use the **Close()** method to close the stream(the connection to the file).

StreamWriter and StreamReader

StreamWriter & StreamReader

- StreamWriter: is a class for writing texts to a Stream by converting characters into bytes.
- StreamReader: is a class for reading characters from a Stream by converting bytes into characters.

StreamWriter & StreamReader

```
//Some of the constructors in StreamWriter class:
//initialize the StreamWriter for the specified FileStream
public StreamWriter(Stream stream);
//initialize the StreamWriter for the specified file path
public StreamWriter(string path);

//Some of the constructors in StreamReader class:
//initialize the StreamReader for the specified FileStream
public StreamReader(Stream stream);
//initialize the StreamReader for the specified file path
public StreamReader(string path);
```

StreamWriter & StreamReader

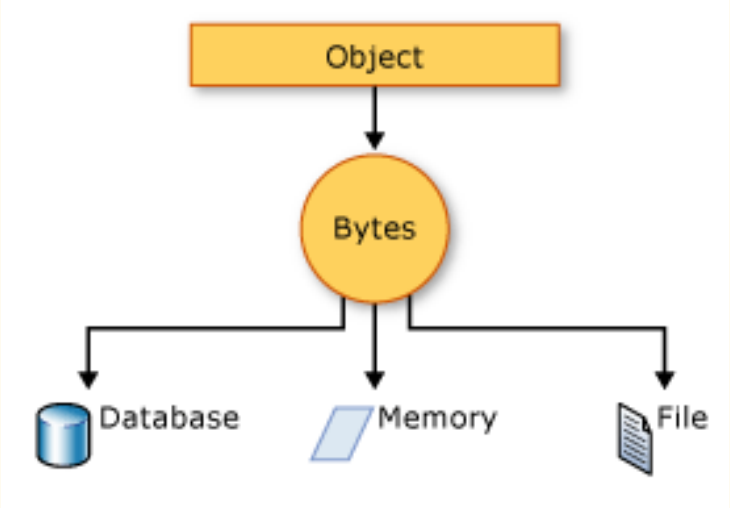
Methods(StreamWriter)	Description
Write(string value)	It writes data to the stream.
WriteLine(string value)	It is the same as Write() but it adds the newline character at the end of the data.
Close()	This method closes the current StreamWriter.

Methods(StreamReader)	Description
ReadLine()	This method **reads** a line of characters from the current stream and returns the data as a string.
Close()	The Close() method **closes** the StreamReader object and the underlying stream, and releases any system resources associated with the reader.

Serialization

Serialization

- Serialization is **the process of converting an object into a stream of bytes** to store the object or transmit it to memory, a database, or a file. Its main purpose is to save the state of an object in order to be able to recreate it when needed. The reverse process is called **deserialization**.



Exception

Exception

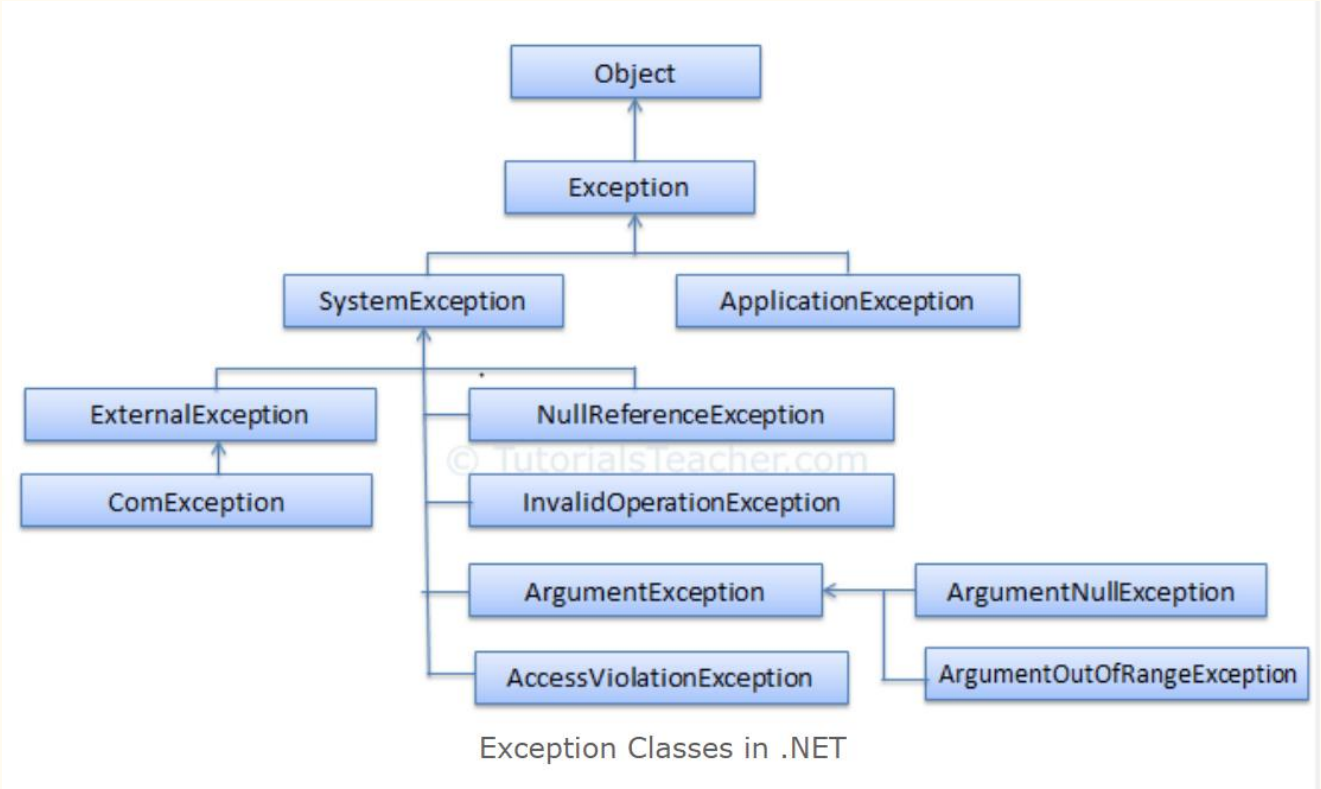
An **exception** is an unwanted or unexpected event, which occurs during the execution of a program i.e at run time, that disrupts the normal flow of the program's instructions. That means the statements placed after the exception-causing statements are not executed but the statements placed before that exception-causing statement are executed by CLR.

Exception class in C# is responsible for exceptions

Exceptions can arise due to a number of situations.

- Trying to access the 11th element of an array when the array contains only 10 elements. (IndexOutOfRangeException)
- Division by zero (DivideByZeroException)
- Accessing a file which is not exist (FileNotFoundException)
- Failure of I/O operations (IOException)

Exception Hierarchy



Exception Classes in .NET

Exception Handling

- try-catch
- finally
- throw

try-catch

- try/catch block can be placed within **any method** that you feel may raise exceptions
- All the statements to be tried for exceptions are put in a try block
- catch block is used to catch any exception raised from the try block
- If an exception occurs in any statement in the try block, the control immediately passes to the corresponding catch block

try-catch

- Order of catch is important
  - Always put smaller exceptions in front of bigger exceptions
  - Don’t put big basket in the front; you are going to catch everything
  - Don’t put small basket in the end; you will not catch anything
- Nested try-catch block is allowed

finally

- By using a ***finally*** block:
  you can **clean up any resources** that are allocated in a try block
  you can **run code even if an exception occurs** in the try block.

- Typically, ① the statements of a finally block run when control leaves a try statement. ② The transfer of control can occur as a result of normal execution, ③ of execution of a return or break statement, ④ or of propagation of an exception out of the try statement.

```
2 references
internal class Program
{
    private StringBuilder sb = new StringBuilder("OgString");

    //Display the string in the StringBuilder
    1 reference
    private void DisplayString()
    {
        Console.WriteLine(sb.ToString());
    }
```

```
program.MethodOne();
program.DisplayString();
```

```
private void MethodOne()
{
    try
    {
        sb.Append(" + TryOne");
        MethodTwo();
    }
    catch (Exception)
    {
        sb.Append(" + CatchOne");
    }
}

1 reference
private void MethodTwo()
{
    try
    {
        throw new ArithmeticException();
    }
    catch (ArithmeticException)
    {
        throw new Exception();
    }
    finally
    {
        sb.Append(" + FinallyTwo");
    }

    sb.Append(" + Message in MethodTwo");
}
```

throw

- Used to explicitly throw an exception
- Useful when we want to throw a user-defined exception.
- The syntax for throw keyword is as follows:
  - throw new ThrowableInstance
    - eg. throw new **`ArgumentNullException`**();

throw

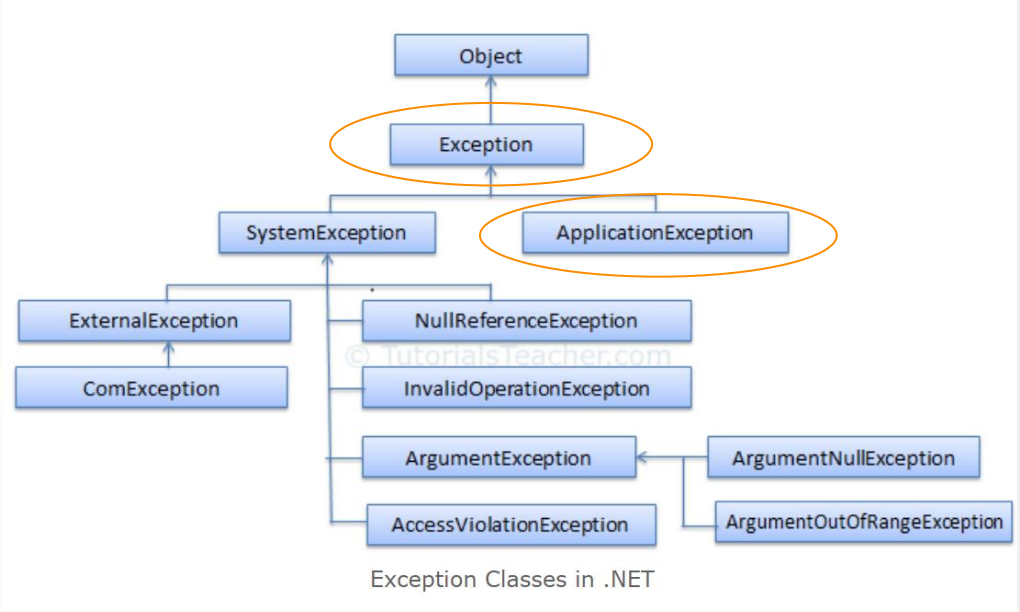
```
static void Main(string[] args)
{
    Student student = null;
    try
    {
        Greeting(student);
    }
    catch (Exception e)
    {

        Console.WriteLine(e.Message);
    }
}
```

```
2 references
class Student
{
    0 references
    public int Id { get; set; }
    1 reference
    public string Name { get; set; }
}
```

```
1 reference
private static void Greeting(Student s)
{
    if (s == null)
    {
        throw new ArgumentNullException();
    }

    Console.WriteLine("Hello " + s.Name);
}
```

User-defined Exception

if none of the predefined exceptions meets your needs, you can create your own exception classes by deriving from the **Exception** class



Exception Classes in .NET

User-defined Exception

- How to create your Exception:
  - Define a new class inheriting from the **Exception** class
  - Override the virtual members that are defined inside the Exception class based on your need;
  - Throw the custom Exception instance where you need it

User-filtered Exception Handlers

User-filtered exception handlers catch and handle exceptions based on requirements that you defined for the exception. This is useful when one catch statement(a particular exception object) corresponds to multiple exceptions;
use the ***catch*** statement with the ***when*** keyword, only when the condition in when(condition) is evaluated as true, the catch statement will get executed.

```
try
{
    //Try statements.
}
catch (Exception ex) when (ex.Message.Contains("404"))
{
    //Catch statements.
}
```

Delegate

Delegate

- A delegate is a container for holding the reference of a method or function, it's described as a "Function Pointer".

- A delegate can be declared using the ***delegate*** keyword, and it can be declared within a class or a namespace(more common).

- The signature of the method must match the signature of the delegate.

```
keyword: delegate
[access_modifier] [delegate] [return_type] [delegate_name] (params);
public delegate void MyDelegate(string msg);

//To use delegate
MyDelegate d;
```

Delegate

There are 3 steps involved while working with delegates:

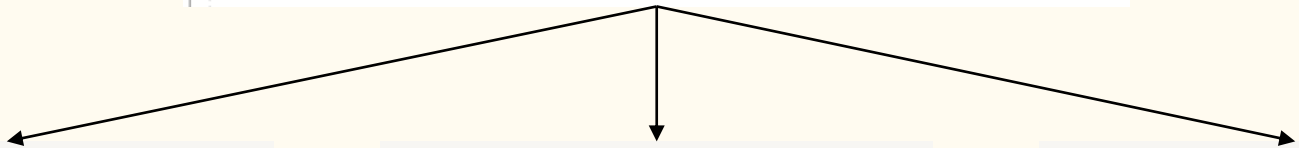
1. Declare a delegate (in a class or namespace)

2. Instantiate a delegate

3. Invoke a delegate

Step1: Declare a Delegate

```
namespace DelegateExamples
{
    //step1: declare
    //declare a delegate with no parameters and no return value
    public delegate void NoParamNoReturnDelegate();
```



```
public void MethodA()
{
    Console.WriteLine("Method A");
}
```

```
public void MethodB()
{
    Console.WriteLine("Method B");
}
```

```
public void MethodC()
{
    Console.WriteLine("Method C");
}
```

Step2: Instantiate a delegate

```
0 references
static void Main(string[] args)
{
    //step 2: set a target method to the delegate
    NoParamNoReturnDelegate d1 = new NoParamNoReturnDelegate(NoParamNoReturnFunction);

    //another way for step 2
    NoParamNoReturnDelegate d2 = NoParamNoReturnFunction;
```

```
3 references
public static void NoParamNoReturnFunction()
{
    Console.WriteLine("No params and return void!");
}
```

Step3: Invoke a delegate

```
0 references
static void Main(string[] args)
{
    //step 2: set a target method to the delegate
    NoParamNoReturnDelegate d1 = new NoParamNoReturnDelegate(NoParamNoReturnFunction);

    //another way for step 2
    NoParamNoReturnDelegate d2 = NoParamNoReturnFunction;
```

```
//step3: invoke a delegate
d1.Invoke();

//another way for step3
d2();
```

Multicast Delegate

- The delegate that points to multiple methods is called a multicast delegate.
- The addition operator adds a function to the invocation list, and the subtraction operator removes it.
- If a delegate returns a value, then the last assigned target method's value will be return when a multicast delegate called.

```
1 reference
public class ClassA
{
    1 reference
    public static void MethodA()
    {
        Console.WriteLine("This is A method");
    }
}
1 reference
public class ClassB
{
    1 reference
    public void MethodB()
    {
        Console.WriteLine("This is B method");
    }
}
```

```
namespace MulticastDelegate
{
    public delegate void MyDelegate();
    0 references
    internal class Program
    {
        0 references
        static void Main(string[] args)
        {
            MyDelegate md1 = ClassA.MethodA;
            MyDelegate md2 = new ClassB().MethodB;
            MyDelegate md3 = md1 + md2;
            md3();

        }
    }
    1 reference
```

Delegate

Delegates are often used as:

- Members of a class
- Parameters of a function/method

Delegate

- (used as member of the class)

```
namespace MyNamespace
{
  public delegate void MyDelegate(int num);
  public class MyClass
  {
    public int myInt;
    public MyDelegate myDelegate;
    .....
  }
}
```

Delegate

- (used as parameters of a function/method)

```
public class Exam
{
    // Output the result based on the student's score
    1 reference
    public void PassExamWithoutDelegate (Student s)
    {
        if (s.Score >= 60)
        {
            Console.WriteLine("Pass! Score = {0}", s.Score);
        }
        else
        {
            Console.WriteLine("Fail! Score = {0}", s.Score);
        }

    }
```

```
// user-defined delegate
public delegate void PassAsParamExampleDelegate(Student student);
10 references
```

```
// Output the result based on the student's score using Delegate
1 reference
public void PassExamWithDelegate
    (PassAsParamExampleDelegate d, Student s)
{
    d(s);
}
```

Build-in Generic Delegates

Func Delegate

- ***Func*** is a generic delegate included in the System namespace. It has zero or more input parameters and one out parameter. The last parameter is considered as an out parameter.
- Func<params_types..., return_type> variableName;

```
//Func<> -- return a result
//Func with 2 input parameters and one return
Func<int, int, int> addNumberDelegate = Sum;
int result = addNumberDelegate(10, 20);
Console.WriteLine(result);

//Func with no input and one return
Func<bool> returnTrueDelegate = ReturnTrue;
Console.WriteLine(returnTrueDelegate());
```

```
static int Sum(int a , int b)
{
    return a + b;
}

1 reference
static bool ReturnTrue ()
{
    return true;
}
```

Action Delegate

- ***Action*** is a delegate type defined in the System namespace. An Action type delegate is the same as ***Func*** delegate except that the Action delegate doesn't return a value. In other words, an Action delegate can be used with a method that has a void return type.
- Action<params_types, ...> variableName;

```
//Action<> -- return void
//Action with 2 input parameters and no return
Action<int, int> printSumDelegate = PrintSum;
printSumDelegate(1, 2);
```

```
1 reference
static void PrintSum(int a, int b)
{
    Console.WriteLine("The result is: " + a + b);
}
```

Predicate Delegate

- A ***predicate*** delegate method must take one input parameter and return bool - true or false.

```
//Predicate<> -- take in one parameter return bool
Predicate<int> isAdultDelegate = IsAdult;
Console.WriteLine(isAdultDelegate(18));
```

```
1 reference
static bool IsAdult( int a)
{
    return a >= 18 ? true : false;
}
```

Lambda Expressions

- Lambda expression in C# is the shorthand for representing anonymous method.

- Syntax:
  - Parameters => Body expression

```
public class Test
{
    0 references
    public int AddTwoNumbers(int num1, int num2)
    {
        return num1 + num2;
    }
}
```

```
//Deelegate without Lambda
Test test = new Test();
Func<int, int, int> addNumber = test.AddTwoNumbers;
addNumber(2, 3);

//Delegate with Lambda
Func<int, int, int> addNumberLambda = (x, y) => x + y;
addNumberLambda(2, 3);
```

Lambda Expression

- Lambda expression can have zero parameter:

```
() => Console.WriteLine("Hello")
```

- Lambda expression can have multiple parameters in parenthesis ():

```
(a, b, c) => a + b - c
```

- Lambda Expression can have multiple statements in body expression in curly brackets {}

```
a =>
{
  Console.WriteLine("Hello");
  return a + 1;
}
```

- Lambda Expression can be assigned to Func, Action or Predicate delegate

```
Func<Student, bool> isStudentTeenager = s => s.Age > 12 && s.Age < 20;
```

Outline

- File Handling (System.IO)
- Exception
- Delegate
- Lambda Expression

Question?